Desarrollo de algoritmos para un Grid de sistemas empotrados

José Rafael Guerra González

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO FIN DE GRADO EN INGENIERÍA DEL SOFTWARE

Curso académico 2016/2017. Convocatoria de junio

Directores

Prof. Dr. Fernando Castro

Dr. José Miguel Montañana

Autorización de difusión y utilización

Se autoriza a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores y directores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Madrid, 16 de Junio de 2017

Fdo. _____

“Sólo podemos ver poco del futuro, pero lo suficiente

para darnos cuenta de que hay mucho que hacer”

Alan Turing (1912-1954)

Agradecimientos

A mis tutores, Fernando Castro Rodríguez y José Miguel Montañana por toda la ayuda brindada a lo largo de mi proyecto fin de grado, en momentos de cálculos erróneos supieron guiarme y darme el suficiente ánimo como para volver a comenzar.

Gracias a mi esposa Hemely y a mi familia por todo el apoyo brindado durante estos 5 años.

Gracias a la Universidad Complutense de Madrid por todos los conocimientos transmitidos que me han ayudado a desarrollar una visión crítica y conseguir las aptitudes necesarias para desarrollar mi carrera profesional en cualquier ámbito de la informática

Resumen

La programación paralela es indiscutiblemente el futuro próximo de las empresas ya que cada día surgen nuevos problemas que deben ser resueltos lo antes posible, esto se puede ver reflejado en la cantidad de programas que trabajan de forma ineficiente (por la cantidad de tiempo que requieren). Es por ello que se necesitaran de proyectos de investigación que intenten solucionar este tipo de problemas para poder hacer un mejor uso de los recursos.

El grid desarrollado funciona en esta línea, ayuda a realizar un estudio sobre programas ejecutados en paralelo (utilizando MPI) sobre una placa (arquitectura Intel) desarrollada para el IoT (Internet of Things) proporcionando estudios estadísticos sobre si es mejor tener un sistema centralizado o más distribuidos.

Todo esto se consiguió gracias a la implementación de diferentes algoritmos (siendo estos ejecutados de forma secuencial y paralelo) los cuales nos permitieron explotar los diferentes recursos hardware en la placa Intel Galileo Gen 2.

Palabras clave: Grid, sistemas distribuidos, rendimiento, paralelismo.

Abstract

Parallel Programming is undoubtedly the future of companies, with new problems arising every day, is necessary to be solved as soon as possible. This can be seen in the number of programs that work inefficiently (for the amounts of time they require in perform its calculations). That's why projects like this will be needed to help solve these kinds of problems to make a better use of the resources.

The grids developed works in this line, it helps to carry out a study on programs executed in parallel (using MPI) on a board (Intel architecture) developed for the IoT (Internet of Things) providing statistical studies on whether it is better to have a system centralized or more distributed.

All this was achieved thanks to the implementation of different algorithms (implemented in sequential and parallel) which allowed us to exploit all the hardware resources on the board Intel Galileo Gen 2.

Keywords: Grid, distributed systems, performance, parallelism,

Índice

Índice de Ilustraciones

Índice de Tablas

Capitulo 1. Introducción

El acelerado crecimiento de internet y la disponibilidad de computadores cada vez más potentes y económicos, han ayudado a cambiar la forma en que los científicos y los ingenieros desarrollan sus cálculos y procesos. Esta nueva tecnología (Internet) y los nuevos recursos cada vez más sofisticados (CPU, memoria, etc.) han permitido poder agrupar una gran cantidad de recursos informáticos y convertirlos en supercomputadores para realizar cálculos cada vez más complejos. Esto a su vez permite al usuario el acceso interactivo a un sinfín de recursos en sistemas distribuidos. A este nuevo paradigma se le denomina **"Grid computing".**

1.1 Grid computing

Según **Ian Foster [1]** , el "**Grid computing** *es un sistema donde se coordinan recursos los cuales no están sujetos a ningún control de forma centralizada y estos a su vez proporcionan unos protocolos e interfaces no triviales que ayudan a proporcionar calidades de servicio"*.

La finalidad del **Grid computing** es la de proveer servicios orientados a infraestructuras y estandarización de protocolos para permitir el fácil acceso al hardware, software y otros recursos. Este desarrollo de tecnología permite a los científicos e ingenieros poder trabajar en un nuevo modelo donde se proporcionan soluciones a problemas cada vez más complejos.

Este cambio no ocurrió de forma repentina, si no que se consiguió gracias a la evolución de la programación distribuida y la programación paralela. Esta corriente se desarrolla a partir de los años 1980 y 1990 donde hubo una gran cantidad de investigaciones y de los que podemos destacar los siguientes: Linda, Concurrent Prolog, BSP, Occam, Fortran-D, C++, pC++, MPI, OpenMP, entre otros. Estos proyectos fueron utilizados para crear nuevos servicios basados en red, creando a su vez un nuevo paradigma en la programación.

Estas nuevas capacidades de cómputo en sistemas distribuidos han permitido que los nuevos cálculos en ordenadores interconectados por red sean mucho más complejos (tanto por el tamaño del problema como la cantidad de recursos que consume) que si los comparásemos con la forma de cálculo en ordenadores tradicionales. Este incremento de recursos informáticos ha ayudado a aprovechar los tiempos muertos en las CPU que existían en los equipos tradicionales, haciendo los actuales miles de veces más potentes.

Una división básica de los tipos de grids podría ser la siguiente:

- **Computational grids:** estos grids dan un acceso seguro a recursos computacionales, donde se tiene potencia suficiente para poder resolver grandes problemas que computacionalmente son complejos.
- **Utility grid:** en este tipo de grids, no sólo los ciclos de la CPU son compartidos, sino también softwares y otros periféricos como sensores.
- **Network grid:** aunque se tuviese un computador con mucha potencia, si fuera parte de un grid con una conexión de red lenta, éste no podría comunicarse de forma óptima que el resto de ordenadores en el grid. Los grids en red ofrecen un alto rendimiento en el intercambio de mensajes ofreciendo conexiones de alta velocidad.
- **Data grid:** es un grid con una gran variedad de servicios informáticos que permiten a una persona o un grupo de usuarios tener acceso, modificar o transferir grandes cantidades de datos.

1.1.1 Arquitectura de un grid

Los recursos pertenecientes a la infraestructura de un grid pueden estar distribuidos en varios nodos y no necesariamente tienen que estar en una misma localización geográfica. Cada uno de estos nodos y recursos pueden ser independiente y estar separado sin generar ningún tipo de problema.

En la siguiente ilustración se explica cómo es el modelo en capas de un grid.

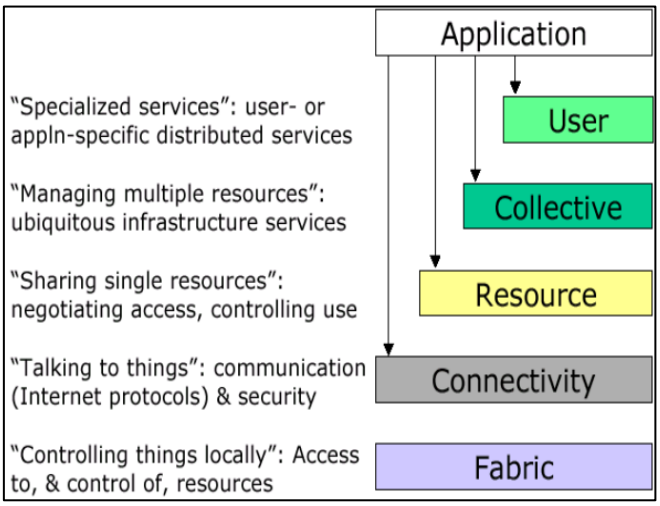


Ilustración 1 *Capas de un grid [1]*

- **Application:** la capa superior representa la aplicación que se va a ejecutar, esta capa provee los mecanismos suficientes para que el usuario pueda interactuar con el grid.
- **Collective:** esta capa lo que provee son utilidades de propósito general, siendo las más destacadas: intercambio de directorios y ficheros, servicios, monitoreo de red y equipos, diagnóstico, replicación de datos, etc.
- **Resource**: en esta capa se definen los protocolos que se podrán utilizar para poder realizar el intercambio de información. Específicamente en esta capa se definen las Aplication Program Interface (API) y Software Development Kit (SDK) para poder realizar conexiones, registró, monitoreo de conexiones realizadas y compartición de recursos.
- **Connectivity**: esta capa se encarga de proporcionar conexiones seguras con un fácil acceso; aquí se tienen protocolos donde se permite el intercambio de datos entre la capa de recursos y la capa de red del modelo OSI. Cuando se habla de conexiones seguras significa que provee una criptografía bastante fuerte y los mecanismos necesarios para autenticar usuarios.
- **Fabric:** en esta capa tenemos recursos compartidos como ancho de banda, tiempos de CPU, memoria, instrumentos científicos como sensores, etc. Los datos son recibidos en esta capa y son directamente transmitido a los otros nodos del grid o pueden ser almacenados en una base de datos sobre el grid.

- **User**: en esta capa se representa la interacción del usuario con el sistema; su única interacción con el grid es la de ejecutar tareas.

1.1.2 Aplicaciones de grid computing

Un grid optimiza la utilización de los recursos, como por ejemplo los ciclos de la CPU, los cuales de otra manera se desperdiciarían; gracias a esto los usuarios pueden tener un extra de recursos para el procesamiento de problemas computacionales cada vez más complejos y tener el mismo nivel de cómputo que un súper computador. En este escenario, el grid computing tiene una gran variedad de usos, tanto para enseñanza académica como para uso científico. A continuación, se nombran algunos de los mayores beneficios de un grid:

- Aporta eficiencia con un coste menor
- Optimiza la utilización de recursos
- Utiliza recursos virtuales y organizaciones virtuales
- Incrementa la capacidad y la productividad
- Mayor balance de los recursos
- Reduce el tiempo de respuesta

El grid computing provee una nueva forma en la que la computación puede resolver problemas financieros, climáticos, simulación de terremotos, etc. de una forma más económica.

1.1.3 Retos en un grid computing

Como se ha mencionado anteriormente, el grid aporta múltiples beneficios, sin embargo puede tener algunas desventajas; un grid soluciona problemas pero trae consigo nuevos retos que los ingenieros tienen que solucionar, por ejemplo, en un super computador no existe el problema del retardo entre las comunicaciones porque todo se hace con los recursos internos del computador (CPU, registros, RAM, etc.). En un grid se tiene que tener en cuenta este factor e intentar dotarlo siempre que se pueda de una

conexión fiable y rápida para no retrasar al sistema. Otra problemática que se puede destacar es la fiabilidad del grid; Para garantizarla se deben de implementar sistemas de monitoreo para poder conocer el estado de los recursos del grid.

También se debe tener en cuenta la disponibilidad de los datos y cómo pueden ser tratados en nuestro grid; se tienen que definir roles de acceso y cómo pueden ser tratados estos datos para evitar problemas.

La lista de retos sobre un grid no se limita a lo mencionado anteriormente, esto son solo unos cuantos retos que tienen consigo y que tienen que ser solucionados.

Para resolver los retos expuestos a estos sistemas se crearon nuevos paradigmas de programación como por ejemplo la programación multihilo. Los nuevos lenguajes desarrollados permitieron a los programadores poder paralelizar trabajo mejorando la productividad del sistema.

1.2 Programación multihilo

En arquitectura de los computadores, un hilo de ejecución fue definido por **Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin** [1] como "*la unidad básica de la utilización de la CPU (Central Processing Unit) donde cada hilo posee su propio contador de programa, cola, registros del sistema y un identificador (Thread ID)".*

Un hilo tiene su propio id, contador de programa y registros del sistema. Los hilos proveen mecanismos para aumentar el rendimiento paralelizando trabajo, reduciendo la sobrecarga que producen los procesos normales. Estos hilos y procesos son utilizados en implementación de servicios de red y servidores web.

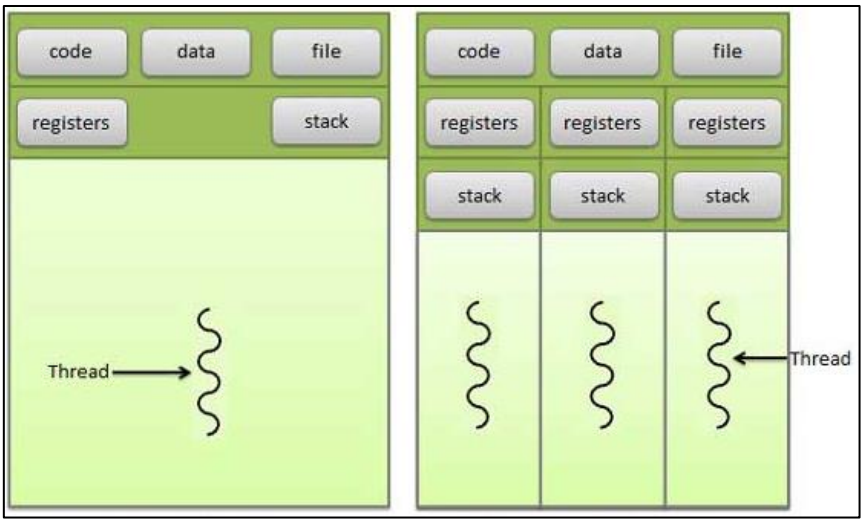


Ilustración 2 Hilos de ejecución [1]

En la tabla 1 se logra apreciar las diferencias entre trabajar con procesos e hilos.

Los hilos pueden ser de dos tipos, los gestionados por el Kernel y los gestionados por el usuario o aplicación. Cada uno de ellos ofrece sus ventajas, a continuación, se detalla brevemente cada uno de estos casos.

Hilos gestionados por el usuario o aplicación: son aquellos que están gestionados por la librería de donde fueron invocados; estos hilos aún necesitan llamadas al sistema para operar, pero esto no significa que el Kernel conozca o controle estos hilos. Estos hilos tienen una pequeña desventaja ya que el Kernel no prioriza estos hilos como sí los hace con los del sistema operativo.

Estos hilos normalmente son bastante rápidos y únicamente se sincronizan entre ellos cuando se hacen una llamada de sincronización. Estos pueden ser una buena elección si estamos frente a un problema que no requiera bloquear tareas para esperar otras (o por cualquier otro motivo). Otra de sus ventajas es que no tiene dependencias del sistema operativo y pueden operar sin ningún problema si cambiamos de sistema, aunque por desgracia muchos de estos hilos suelen ser bloqueados por el sistema operativo para dar prioridad a los hilos del Kernel.

Hilos gestionados por el Kernel: Son una buena elección cuando tenemos que hacer tareas que habitualmente son bloqueadas, aunque si alguno de estos hilos es bloqueado, el proceso general no se ve afectado por esto.

Procesos	Hilos
Son considerados procesos pesados y consumen una gran cantidad de recursos del sistema	Son denominados procesos ligeros y consumen muchos menos recursos que un proceso normal
Los procesos necesitan interactuar con el sistema operativo	Los hilos no necesitan interactuar con el sistema operativo
Cuando se trabaja con múltiples procesos, cada proceso ejecuta el mismo código pero utiliza su propia memoria y registros.	Puede compartir recursos como archivos abiertos.
Si un proceso es bloqueado, los procesos hijos no pueden continuar su ejecución hasta que el proceso padre sea desbloqueado	Si un hilo es bloqueado, otros hilos con la misma tarea puede continuar su trabajo sin problemas
Si se trabaja con procesos sin utilizar hilos, estos terminan usando más recursos	Múltiples hilos consumen mucho menos recursos
En la ejecución de múltiples procesos, cada uno de ellos es independiente de los demás	Un hilo puede leer o escribir los datos de otro hilos

Tabla 1 Comparativa Procesos-hilos.

Estos hilos suelen ser más lentos que los hilos de usuario o aplicación ya que son gestionados por el sistema operativo. Esto se debe al cambio de contexto que necesita más recursos que con los hilos de usuario. Otra de sus desventajas es que no son portables ya que son dependientes del sistema operativo.

Esta nueva forma de trabajar con los recursos del computador dio lugar a dos nuevos lenguajes de programación; el primero denominado openMP [13] (Open Multi-Processing), que trabaja sobre una memoria compartida entre diferentes procesos aprovechando las ventajas de la programación con hilos. El segundo denominado MPI [12] (Message Passing Interface), que nos permite trabajar en un sistema de memoria

distribuida. A continuación se detallan las principales características de ambos paradigmas de programación.

1.2.1 Open Multi-Processing

Open Multi-Processing (openMP) es una API (Application Programming Interface) soportada por diferentes plataformas (Solaris, Linux, OS X, Windows, etc.) que nos permite trabajar con múltiples procesos compartiendo memoria.

OpenMP trabaja generando hilos esclavos [13] con sus variables de entorno cargadas en otro procesador. Cada hilo esclavo lleva asociado su propio identificador siendo éste superior a 0 (0 es el identificador del hilo master); una vez finalizada la tarea, el hilo esclavo se une al hilo master para continuar con la ejecución del programa.

Por defecto cada hilo ejecuta una sección independiente de las otras, siendo el método *Work-sharing constructor* el que nos permite dividir secciones de código entre los diferentes hilos. Esto a su vez nos permite cargar las variables de entorno dependiendo del uso que se le vaya a dar, normalmente las variables en tiempo de ejecución son asignadas a diferentes procesadores dependiendo del hilos que las esté utilizando.

A continuación hablaremos un poco de la evolución histórica de openMP [13]

- OpenMP fue publicado por primera vez en Fortran 1.0 en Octubre de 1997.
- En octubre de 1998 fue publicado el estándar para C/C++ 1.0.
- La versión 2.0 para Fortran y C/C++ fue publicada a comienzos del 2002, esta versión especificaba cómo paralelizar bucles principalmente (programas orientados a cálculos numéricos en matrices).
- La versión 2.5 fue una combinación entre las especificaciones de Fortran y C/C++, siendo ésta publicada en 2005.
- La versión 3.0 fue publicada en mayo del 2008 mejorando las características de la versión 2.0, permitiendo un mejor control y paralelismo entre bucles.
- La versión 4.0 fue publicada en julio de 2013 mejorando las siguientes características: control de errores, hilos affinity, extensión de tareas basadas en hilos y aceleración hardware.

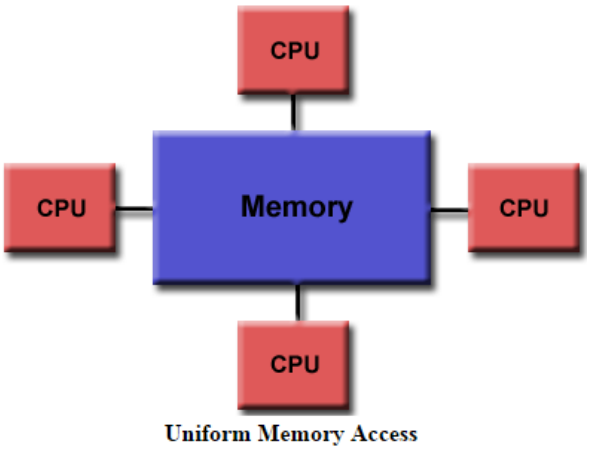


Ilustración 3 Modelo de programación OpenMP [13].

Una de las limitaciones más importantes de OpenMP es que se ve acotado por el número de procesadores en el computador, lo cual provoca que el nivel de paralelismo se vea afectado. Otra limitación importante es que no permite ejecutar código en un sistema de memoria distribuida, cosa que con MPI sí se puede conseguir.

1.2.2 Message Passing Interface

Message Passing Interface (MPI) es un estándar definido por **Poonam Dabas y Annopa Arya [14]** como "una sintaxis y una semántica de funciones diseñada para poder sacar mejor provecho a los múltiples procesadores de un computador". Es una técnica empleada en la programación concurrente que ayuda en la sincronización entre procesos y permite exclusión mutua. Es un protocolo de comunicaciones entre computadores donde los nodos ejecutan programas en un sistema de memoria distribuida. La implementación de MPI consiste en varias bibliotecas que pueden ser utilizadas en lenguajes de programación como C, C++, etc.

Al comienzo de la ejecución del programa se asigna el número de procesos que son requeridos para su ejecución y estos no crean procesos adicionales hasta que termine su ejecución. A cada proceso se le asigna una variable denominada Rank que identifica de manera única a cada proceso. Para controlar la ejecución del programa se utilizan estas variables.

Existen varios tipos de llamadas en MPI [12], tales como:

- Llamadas utilizadas para inicializar, administrar y finalizar las comunicaciones.
- Llamadas utilizadas para transferir datos entre los distintos procesos.
- Llamadas utilizadas para crear diferentes tipos de datos es un proceso concreto.

Para simplificar, se muestran una serie de ventajas de programar en MPI frente a otros lenguajes de programación:

- A diferencia de muchos lenguajes, MPI es la única librería que puede ser considerada en sí misma un estándar.
- Es portable ya que no se necesita modificar el código fuente cuando se cambia de plataforma.
- Intenta mejorar el rendimiento ya que se intenta explotar al máximo las capacidades hardware del computador.
- Existen alrededor de 430 rutinas definidas en MPI-3.
- Está disponible en una gran variedad de implementaciones.

Modelos de programación de MPI

Originalmente MPI fue diseñado para trabajar bajo una arquitectura de memoria distribuida (como se muestra en la ilustración 4) lo cual lo hizo realmente popular en los años 80 y principios de los 90 [12].

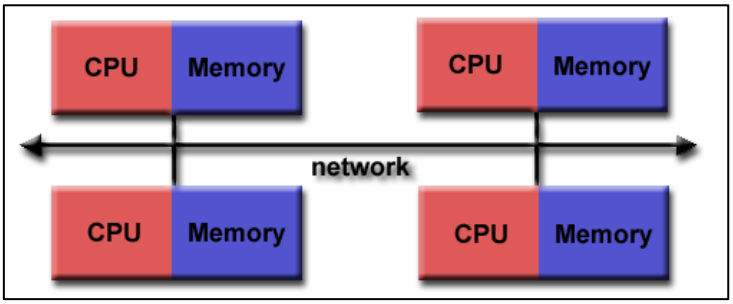


Ilustración 4 Modelo de programación de MPI [12].

Pero con el paso de los años, la arquitectura de los computadores cambió [2] y dio paso a una nueva forma de trabajar, generando un nuevo modelo donde se utilizaban memorias compartidas sobre redes creando un híbrido entre memoria distribuida y memoria compartida del sistema (ilustración 5).

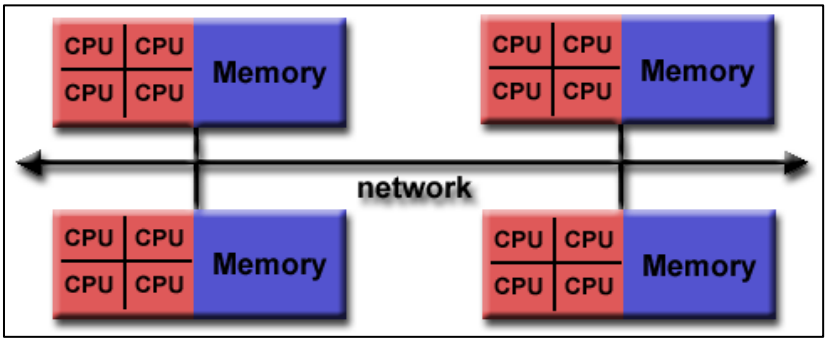


Ilustración 5 Nuevos modelos de programación de MPI [12].

Gracias a estos avances, en la actualidad se trabaja sobre cualquier plataforma, ya sea virtual o física, creando así una nueva arquitectura de trabajo basada en memoria distribuida, memoria compartida y una arquitectura híbrida.

A continuación, se detalla brevemente la evolución histórica de MPI [12]:

- En el verano de 1991 un pequeño grupo de investigadores comenzaron a discutir sobre las características que debería de tener el nuevo estándar de MPI en Austria.
- En abril de 1992, en el centro de investigación para el cómputo paralelo, Williamsburg, Virginia, las características esenciales de MPI fueron discutidas y un grupo de trabajo estableció cómo sería el proceso de estandarización.
- En noviembre de 1992, un grupo de trabajo conocido como Minneapolis, presentaron lo que se llamó MPI-1.
- En noviembre de 1993 se presentó un borrador del estándar MPI.
- En mayo de 1994 se presentó la versión final de MPI1.
- MPI-1.0 fue actualizado a la versión MPI-1.1 en junio del 1995.
- MPI-1.2 fue actualizado en julio del 1997.
- MPI-1.3 fue actualizado en mayo del 2008.

- Se finalizó MPI-2 en 1996, donde se abordaron temas que nunca fueron definidos en las especificaciones para MPI-1.
- MPI-2.1 fue actualizado en septiembre de 2008.
- MPI-2.2 fue actualizado en septiembre de 2009.
- MPI-3 fue creado en septiembre de 2012.
- Y finalmente se llegó a la versión MPI-3.1 publicado el 4 de junio del 2015, que es la versión más actual y con la que se trabaja hoy en día.

A continuación, se ilustra todos los proyectos que ayudaron a la creación de MPI-3.

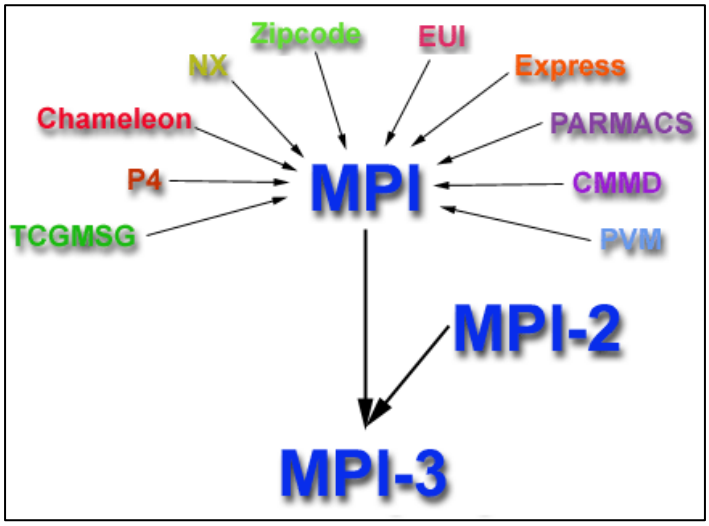


Ilustración 6 MPI [12].

Gracias a herramientas como MPI, programas ejecutados en red pueden ser utilizados en sistemas distribuidos, como por ejemplo en el internet de las cosas (Internet of Things, IoT).

El Internet de las cosas es definido por ***Alberto Tejero López*** [10] como un nuevo paradigma que permite a objetos conectarse e interactuar entre ellos a través de internet. También lo definen como el punto en el tiempo donde habrá más conexiones entre objetos que entre personas [7].

En este nuevo paradigma (IoT) [10], MPI puede ser de mucha utilidad ya que es una herramienta que intenta facilitar la comunicación entre diferentes elementos u objetos en la IoT.

En la sección 1.3 se detalla qué es el Internet de las cosas. Aunque el internet de las cosas no entra dentro del alcance del proyecto, se consideró oportuno hablar de él ya que el IoT ha adquirido mucha importancia en la actualidad; En el futuro se necesitarán proyectos donde se estudie el rendimientos de diferentes placas diseñadas para el IoT y este proyecto puede ser de mucha ayuda.

1.3 Internet of Things

El Internet de las Cosas o IoT (Internet of Things) se constituye como un nuevo paradigma que habilita un sinfín de nuevos servicios y aplicaciones para muchos de los ámbitos de nuestra sociedad [10]. Sin embargo, su seguridad, un punto de especial trascendencia, sigue sin estar del todo clara, lo que supone por un lado un foco importante de riesgos y problemas, pero que al mismo tiempo abre la posibilidad para la generación de nuevas ideas y soluciones.

El propósito de IoT es el de proveer conectividad entre los diversos sistemas, servicios y equipos, esto no es más que la evolución del uso de internet y donde empresas tecnológicas como *Gartner Inc* a través de estudios estadísticos especulan que para 2020 puede haber un total de 26 mil millones de dispositivos conectados a IoT[17]; un ejemplo de lo que se puede hacer sería el de tener una casa Inteligente [19] donde todo se pudiera monitorizar desde internet para controlar los diferentes dispositivos remotamente.

Siguiendo el ejemplo anterior y usando la idea de la ilustración número 7 [10]. Un operario pudiera conectarse a diferentes placas (como la usada en nuestra propuesta, la Intel galileo Gen 2) desde internet a un hogar y controlar la temperatura del hogar remotamente.

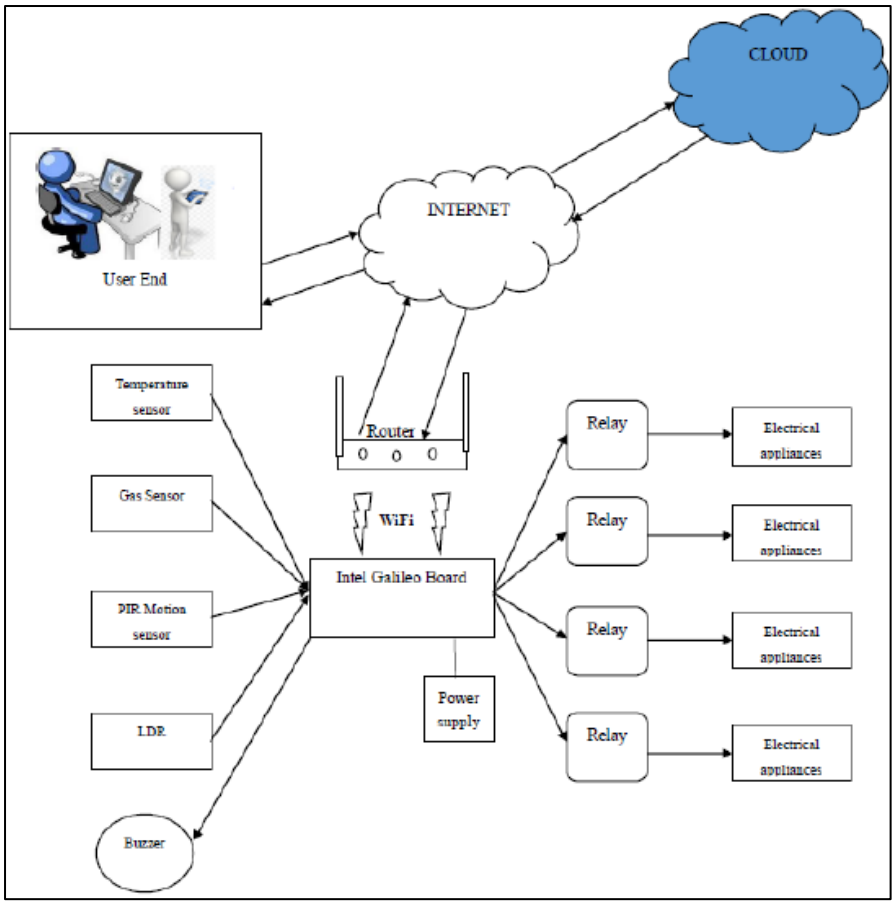


Ilustración 7 Internet of Things con la placa Intel Galileo [10].

El IoT tiene diferentes campos en los que se podría implementar; La siguiente lista intenta dar una idea general de los posibles campos a estudiar:

- Smart Home
- Robótica
- Aviación
- Automatización de las industrias
- Biomedicina
- Monitoreo de fabricas
- Telemetría
- Smart Grids y Smart City’s
- Telemática
- Militar
- Deporte
- Detección de fraude

- Sistemas de transporte
- Climatología
- Transporte (coches, autobuses, motos)
- Bomberos (incendios, sensores de calor)

En la ilustración 8 se exponen las ideas planteadas anteriormente. El IoT junto con las diferentes placas diseñadas para él, nos permitirá controlar diferentes elementos de nuestra vida remotamente mejorando la eficiencia y el ahorro de recursos.

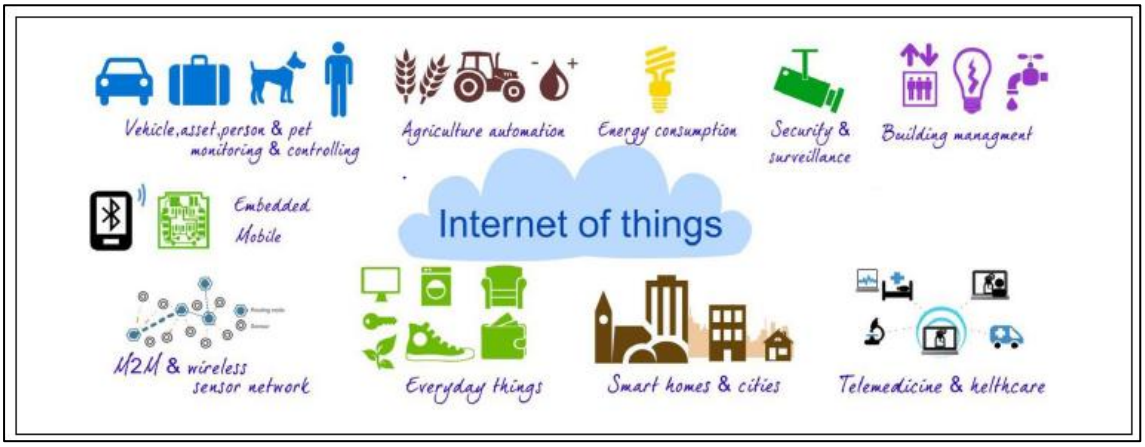


Ilustración 8 Casos de uso de Internet of things [10]

1.3.1 Expectativas futuras

De acuerdo con *Gartner Inc.*, en 2020 habrá alrededor de 26 mil millones de equipos conectados al IoT [17], por lo que se tiene que preparar toda la infraestructura necesaria para poder dar servicio a todos estos nuevos sistemas.

Muchos expertos aseguran que el IoT cambiará de forma radical la forma en que se vive hoy en día [10]; por ejemplo, los nuevos dispositivos Inteligentes serán capaces de ahorrar energía (tanto a nivel local como por ejemplo una casa o de forma más general como una ciudad).

Al integrar estos nuevos dispositivos a internet surge una serie de problemas que vale la pena recalcar, entre los principales podemos resaltar el uso de direccionamiento IP basado en IPv4 donde solo permite utilizar 2^{32} direcciones, con lo cual se plantea que IoT trabaje sobre el protocolo IPv6 ya que este permite trabajar con 2^{128} direcciones IP.

Otro de los inconvenientes que se plantean es la cantidad de datos que se tienen que transmitir y analizar entre los diferentes dispositivos interconectados en la IoT, por lo que no se pueden tratar de la manera tradicional ya que estos retrasarían mucho las comunicaciones, por lo que se tendrá que llegar a una solución en la que se puedan trabajar con herramientas de big-data en sistemas distribuidos.

Además de lo previamente mencionado, queda el aspecto más importante dentro de la IoT que es la seguridad, está fue definida por el profesor *Alberto Tejero López* [10] de la universidad politécnica de Madrid como "*en su aspecto más técnico, se puede definir como aquellas actividades enfocadas a proteger un determinado dispositivo o servicio, así como todo aquello con lo que interactúa e intercambia con otros dispositivos o servicios, ya sea información, datos, señales, etc. Utilizando como base la anterior definición, la seguridad de IoT se podría definir como aquellas actividades encaminadas a la protección de los objetos y sus comunicaciones o interacciones con otros objetos".*

Ante este nuevo escenario tecnológico, se deberán ampliar los estudios e investigaciones al respecto, siendo uno de ellos nuestra propuesta, donde se evalúa el rendimiento de la placa *Intel Galileo Gen 2* frente a algoritmos bien conocidos.

1.4 Objetivos del presente TFG.

El objetivo principal del proyecto es desarrollar un grid computing sobre la placa Intel *Galileo Gen 2* con la finalidad de estudiar el comportamiento de ciertos algoritmos. Estos algoritmos (con distinto grado de paralelismo y diferentes requerimientos de recursos del sistema) fueron programados de forma secuencial y en paralelo para poder estudiar su tiempo de ejecución.

Objetivos Generales

- Búsqueda de información sobre Grid Computing, openMP y MPI.
- Estudio de diferentes algoritmos.
- Buscar publicaciones científicas relacionadas con nuestra propuesta.

Objetivos Específicos

- Definir y explicar los términos Grid Computing y MPI.
- Estudiar las características básicas de los sistemas empotrados.
- Implementación de los algoritmos estudiados de forma secuencial y en paralelo.
- Definir las variables de rendimientos de los sistemas de programación paralela.
- Estudiar y evaluar diferentes algoritmos con diferentes grados de paralelismo.
- Analizar y comparar resultados de publicaciones científicas a los resultados obtenidos por nuestra propuesta.

1.5 Plan de trabajo

ETAPAS	Octubre 2016	Noviembre 2016	Diciembre 2016	Enero 2017	Febrero 2017	Marzo 2017	Abril 2017	Mayo 2017	Junio 2017
Investigación y búsqueda teórica de Grid Computing y MPI									
Estudio bibliográfico de sistemas empotrados y programación en paralelo									
Redacción del marco teórico									
Desarrollo e implementación de la experimentación y obtención de resultados									
Analizar los resultados obtenidos teniendo en cuenta las variables planteadas									
Comparar los resultados con el marco teórico									
Redacción de conclusiones de la investigación.									

Tabla 2 Plan de trabajo

Chapter 1 Introduction

The accelerated growth of the internet and the availability of more powerful and economical computers have helped to change the way that scientists and engineers develop their calculations and processes. These new technologies (Internet, etc.) and resources (CPU, memory, etc.) have made it possible to group up a large amount of computing resources and convert them into supercomputers to perform increasingly complex calculations. This allows the user to get access endless resources in distributed systems. This new paradigm is called "Grid computing".

1.1 Grid computing

According to Ian Foster [1], "Grid computing is a system where coordinate resources which are not centrally controlled and which in turn provide non-trivial protocols and interfaces that help deliver service quality."

The purpose of Grid computing is to provide infrastructure-oriented services and standardization of protocols to allow easy access to hardware, software and other resources. This development of technology allows scientists and engineers work with a new model where solutions to increasingly complex problems are provided.

This change did not occur abrupt way, but it was achieved thanks to the evolution of distributed programming and parallel programming. This trend of develops from the 1980s and 1990s where there was a lots of research and we can highlight the following: Linda, Concurrent Prolog, BSP, Occam, Fortran-D, C ++, pC ++, MPI, OpenMP, and others. These projects were used to create new network-based services, in turn to creating a new paradigm in programming.

These new computing capacities in the distributed systems have allowed new computations in networked computers to be much more complex (both because of the size of the problem and the amount of resources consumed) than if we compare them with the traditional computers. This increase in computing resources has helped to take advantage of downtime in the CPUs that existed in traditional computers, making the current thousands of times more powerful.

A basic sub-division of grids types could be as follows:

- **Computational grids:** These grids give secure access to computational resources, where they have enough power to solve large problems that are computationally complex.
- **Utility grid:** In this type of grids, not only CPU cycles are shared, but also software's and other peripherals such as sensors.
- **Network grid:** Even if the grid had lots of very powerful computer, if it were part of a grid with a slow network connection, it could not communicate optimally with the other computers on the grid. Network grids offer high-performance messaging through high-speed connections.
- **Data grid:** Is a grid with a variety of computer services that allow a person or a group of users to access, modify or transfer large amounts of data.

1.1.1 Architecture of a grid

The resources belonging to the infrastructure of a grid can be distributed in several nodes and do not necessarily have to be in the same geographical location. Each of these nodes and resources can be independent and be separated without generating any type of problem.

The following picture explains how the layered model of a grid is.

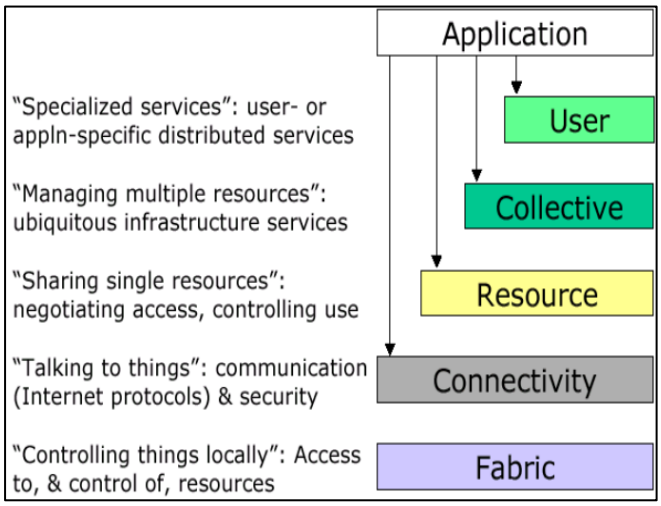


Figure 1 Layers of a grid [1]

- **Application:** The top layer represents the application that will be executed; this layer provides enough mechanisms for the user to interact with the grid.
- **Collective:** This layer provides general purpose utilities, the most prominent being: directory and file sharing, services, network and equipment monitoring, diagnostics, data replication, and so on.
- **Resource:** This layer defines the protocols that can be used to enable the exchange of information. Specifically in this layer the Application Program Interface (API) and Software Development Kit (SDK) are defined to be able to make connections, register, monitor connections and share resources.
- **Connectivity:** This layer is responsible for providing secure connections with easy access; furthermore it has protocols where allow the exchange of data between the resource layer and the network layer of the OSI model. When it comes to secure connections, it means that it provides fairly strong cryptography and the necessary mechanisms to authenticate users.
- **Fabric:** In this layer the grid shared resources like bandwidth, CPU times, memory, scientific instruments like sensors, etc. The data is received in this layer and is directly transmitted to the other nodes of the grid or can be stored in a database on the grid.
- **User:** This layer represents the interaction of the user with the system; His only interaction with the grid is to execute tasks.

1.1.2 Grid computing applications

A grid optimizes the use of resources, such as CPU cycles, which otherwise would be wasted; Thanks to this the users can use an extra of resources for the processing of computational problems increasingly complex and to have the same level of computation that a super computer. In this scenario, grid computing has a wide variety of uses, both for academic teaching and for scientific use. Here are some of the biggest benefits of a grid:

- Provides efficiency at lower cost.
- Optimize resource utilization.
- Uses virtual resources and virtual organizations.
- Increases capacity and productivity.
- Greater balance of resources.
- Reduces response time.

Grid computing provides a new way in which computing can solve financial problems, climate problems, earthquake simulation, etc. In a more economical way.

1.1.3 Challenge of a grid computing

As mentioned above, the grid brings multiple benefits; however it can have some disadvantages; A grid solves problems but brings new challenges that engineers have to solve, for example, in a super computer there is no problem with the delay of the communications because everything is done with the internal resources of the computer (CPU, registers, RAM, etc.). In a grid it must consider this factor and try to equip it with a reliable and fast connection. Another problem that can be highlighted is the reliability of the grid; to guarantee it, monitoring systems must be implemented in order to know the state of grid resources.

It should also consider the availability of data and how they can be treated in the grid, must be defining the access roles necessary and how these data can be treated to avoid problems.

The list of challenges on a grid is not limited to what is mentioned above, these are just a few challenges that have to be considered and have to be solved.

To solve the challenges exposed above, new programming paradigms were created, such as multi-thread programming. The new developed languages allowed programmers to be able to parallelize work improving the productivity of the system.

1.2 Multithread programming

In computer architecture, a thread was defined by ***Abraham Silberschatz, Greg Cagne and Peter Baer Galvin*** [1] as "the basic unit of the CPU (Central Processing Unit) where each thread have its own counter program, system queue, system logs and thread identifier (Thread ID)".

The thread provides mechanisms to increase performance by paralleling work reducing the overhead of normal processes. These threads and processes are used in the implementation of network services and web servers.

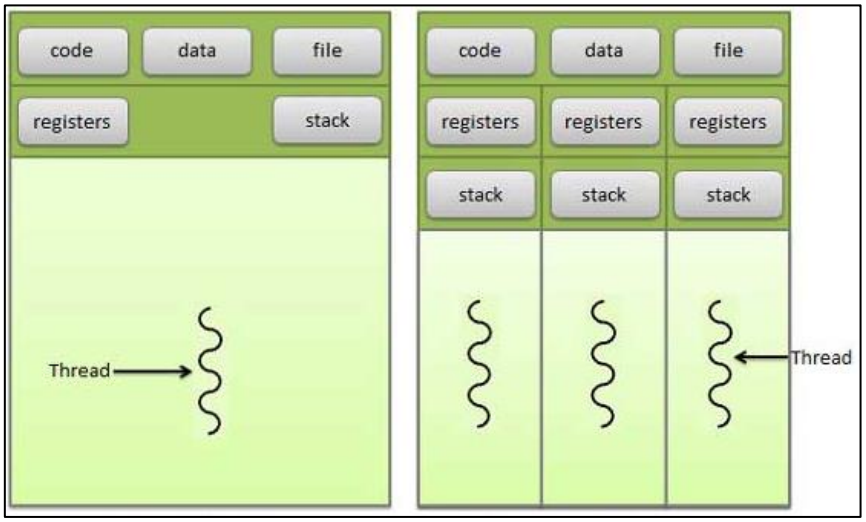


Figure 2 Threads [1].

In the chart 1 it's possible to appreciate the differences between processes and threads:

The threads can be of two types, those managed by the kernel and those managed by the user or application. Each of then offers its advantage:

Threads managed by user or application: are those that are managed by the library where they were invoked; these threads still require system calls to operate but this does not mean that the Kernel knows or controls these threads. These threads have a small disadvantage since the kernel does not prioritizes these threads as it does with its own threads.

These threads are usually quite fast and only synchronize with each other when a synchronization call is made. These can be a good choice if we are facing a problem that does not require blocking task to wait for others threads (or for any other reason). Another advantage is that it has no dependencies on the operative system and can operate without problems if we change the system. Although unfortunately many of these threads are usually blocked by the operating system to prioritizes the kernel thread.

Threads managed by kernel: they are a good choice when it needed to do tasks that are usually blocked, although if any of these threads is blocked, the overall process is not affected by this.

These threads are typically slower than user or application threads because they are managed by the operative system. This is due by the context switch that needs more resources than user threads; another disadvantage is that they are not portable because they are dependent of the operating systems.

Procesess	Threads
They are considered heavy processes and consume a large amount of system resources.	They are called light processes and consume far fewer resources than a normal process.
Processes need to interact with the operating system.	The threads do not need to interact with the operating system.
When working with multiple processes, each process runs the same code but uses its own memory and registers.	It can share resources as open files.
If a process is blocked, child processes cannot continue execution until the parent process is unlocked.	If a thread is blocked, other threads with the same task can continue its work without problems.
If the system works with processes without using threads, they end up using more resources.	Multiple threads consume much less resources.
In the execution of multiple processes, each of them is independent of the others.	A thread can read or write data from other threads.

Chart 1

This new way of working with the resources of the computer gave rise to two new programming languages; the first one is called openMP [13] (Open Multi-Processing), which works on a shared memory between different processes taking advantage of the programming with threads. The second one is called MPI [12] (Message Passing Interface), which allows us to work on a distributed memory system. The main features of both programming paradigms are detailed below.

1.2.1 Open Multi-Processing

Open Multi-Processing (openMP) is an API (Application Programming Interface) supported by different operative systems (such as Solaris, Linux, OS X, Windows, etc.) which allowed to work with multiple processes sharing memory.

OpenMP work generating slave threads [13] with its own environment variable. Each slave threads have associated its own identifier being greater than 0 (0 is the identifier for the thread master); once the task is completed, the slave thread joined to the master thread to continue the execution of the main program.

By default each thread execute a separate section of the program to avoid generating conflict, the work-sharing constructor is the method that allow us to split sections of code between different threads, furthermore, this method allows to load the environment variables depending on the use to be given, normally the variable at runtime are assigned to different processor depending on the threads that are using then.

As following we show a brief history evolution about openMP [13]:

- OpenMP was published in Fortran 1.0 in October 1997.
- In October of 1998 the standard was published for C and C++ 1.0
- Version 2.0 for Fortran and C/C++ was released in early 2002, this version specified how to parallelize loops mainly (Programs oriented to numeric calculations in arrays).
- The version 2.5 was a combination of Fortran and C/C++ specifications, which was published in 2005.
- Version 3.0 was released in May 2008, improving the feature of versions 2.0 allowing better control and parallelism between loops.
- Version 4 was released in july 2013 improving the following features: error handing, affinity threads, threads-based task extension and hardware acceleration.

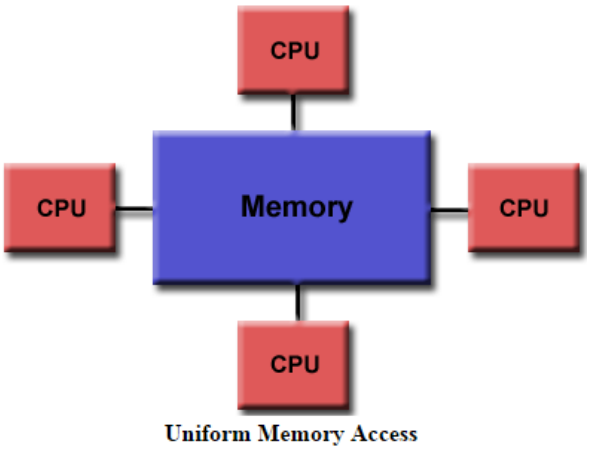


Figure 3 Programing model of OpenMP [13].

One of the most important limitations of OpenMP is that is limited by the number of processors in the computer, which cause the level of parallelism to be affected. Another important limitation is that no allow the executed code in a system of distributed memory, something that with MPI can be done.

1.2.2 Message Passing Interface

Message Passing Interface (MPI) is a standard defined by ***Poonam Dabas and Annopa Arya*** [1] as "a syntax and semantic of functions designed to take advantage of the multiple processors of the computer". It is a technique used in concurrent programming that helps in the synchronization between processes and allows mutual exclusion. It's a communication protocol between computers where nodes execute programs in a distributed memory system. The MPI implementation consists of several libraries that can be used in programming languages such as C, C++ or Fortran.

At the beginning the execution of the program, the number of processes that are require for its execution are assigned and these do not create additional processes until it's execution finishes. Each Process is assigned a variable called Rank that is a uniquely identifies for each process. To control the execution of the program, these variables are used.

They are several types of calls in MPI [12], such as:

- Signals that allow the initialization, managing the session and end of the communications.
- Signals used to transfer data between different processes.
- Signals used to specify the data type for a specific process.

As following we show a series of advantages of programming in MPI against other programming languages:

- Unlike many languages, MPI is the only library that can be considered by itself a standard.
- It's portable because we do not need to modify the source code when we changing the platforms.
- MPI improve the performance trying to fully exploit the computer's hardware capabilities.
- There are about 430 functions defined in MPI-3.
- Available in a variety of implementations.

MPI Programming Models

Originally MPI was designed to work under a distributed system memory architecture (as show in figure 3) which made it really popular in the 1980s and early 1990s [12].

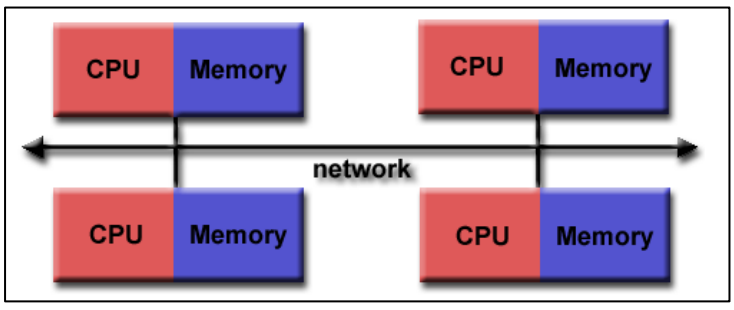


Figure 4 MPI Programming Models [12]

But over the years, the architecture of computers changed [2] generating a new way of working, in this new model where shared memory were used over networks created a hybrid between distributed memory and shared system memory.

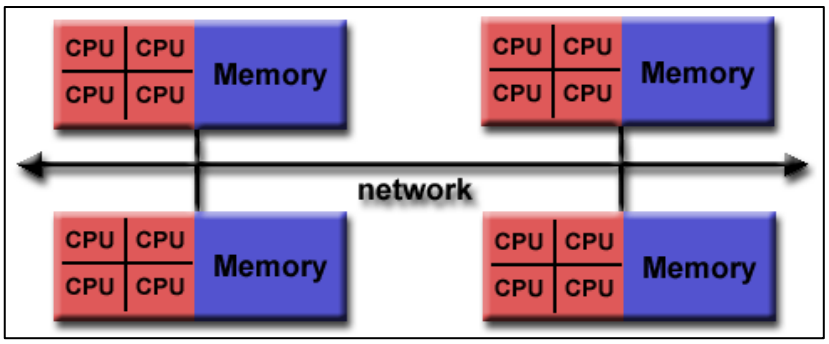


Figure 5 New programming models in MPI [12]

Thanks to these advances, MPI currently working over any platform, even if it's are virtual or physical, creating new work architecture based on distributed memory, shared memory and a hybrid architecture.

As following we show a brief history evolution about MPI [12]:

- In the summer of 1991 a small group of researchers began to discuss the features that should have the new MPI standard in Austria.
- In April 1992, at the parallel computing research center, Williamsburg, Virginia, the essential features of MPI were discussed and a working group established how to standardization process would be.
- In November 1992, a working group knows as Minneapolis, introduced that was called MPI-1.
- In November 1993, a conference on supercomputer was held, where a draft of the MPI standard was presented.
- In May 1994, the final version of MPI-1 was presented.
- MPI-1.0 was upgraded to the MPI-1.1 version in June 1995.
- MPI-1.2 was updated in July 1997.
- MPI-1.3 was updated in May 2008.

- MPI-2 was finished in 1996, where issues that were never defined in the specification for MPI-1 were addressed.
- MPI-2.1 was updated in September 2008.
- MPI-2.2 was updated in September 2009.
- MPI-3 was created in September 2012.
- And finally the version MPI-3.1 was published on June 4, 2015.

As following we show all the projects that helped in the creation of MPI-3:

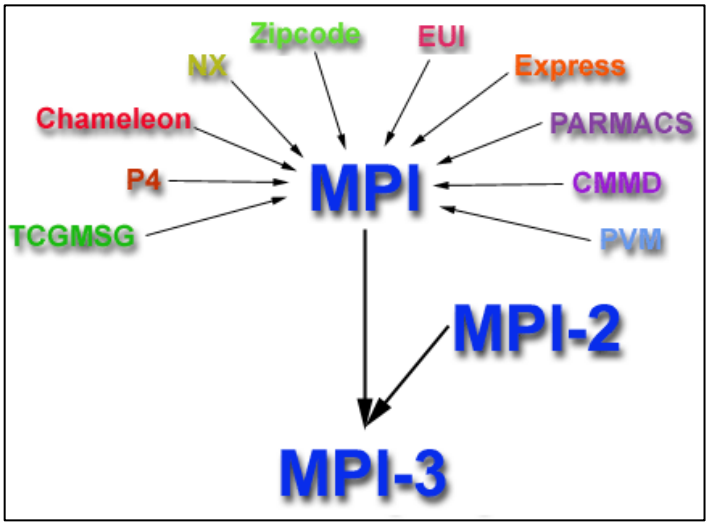


Figure 6 MPI [12]

Thanks to tools like MPI, programs executed in networks can be used in distributed systems, as for example Internet of Things (IoT).

The internet of Things was defined by Alberto Tejero Lopez [10] as a new paradigm that allows objects to connect and interact with each other through the internet. They were defined as the point in time where there will be more connections between objects that between humans [7].

In this new paradigm (IoT) [10], MPI can be very useful since it is a tool that tries to facilities the communication between different elements or objects in the IoT.

Section 1.3 details what is the Internet of Things. Although the IoT does not fall within the scope of the project, it was considered appropriate to talk about it since the IoT has acquired great importance at the present time. In the future will be needed projects that study the performance of different boards designed for IoT and this project can be very helpful since the Intel Galileo Gen 2 board is the one used to evaluate our proposal.

1.3 Internet of Things

The Internet of Things or IoT constitute a new paradigm that enable endless new service and applications for many of the areas of our society [10], however, its security is a point of special importance, is still no completely clear which one is the source of risks and problems, but at the same time opens up the possibilities for generating new ideas and solutions.

The purpose of the IoT is to provide connectivity between the various systems, services and equipment's, this is nothing more than the evolution of internet use and where technology companies such as *Gartner Inc.* through statistical studies speculated that by 2020 there may be a total of 26 billion devices connected to IoT [17]; An example of what can be done would be to have a smart home [19] where everything could be monitored from the internet to control the different devices remotely.

Following the example above and using the idea showed in figure number 7 [10]. An operator could connect to different boards (like the one used in our proposal, The Intel Galileo Gen 2) from the internet to a home and remotely control the temperature.

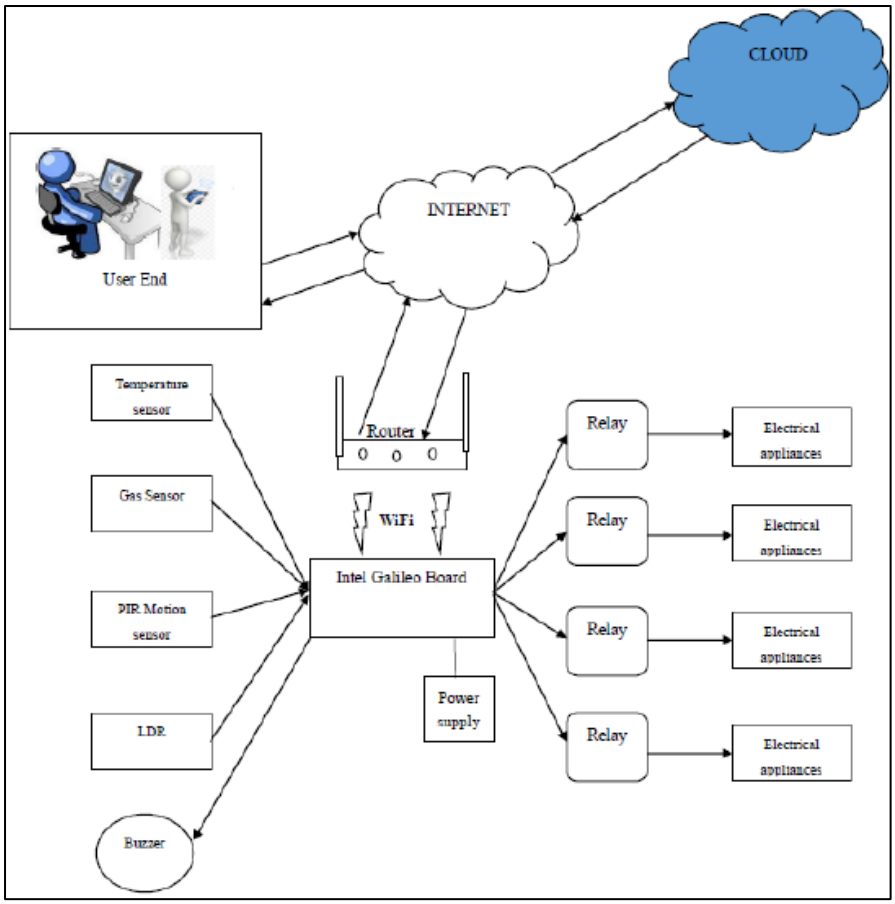


Figure 7 Internet of Things with the Intel Galileo Gen 2 board [10].

The IoT has different fields in which it could be implemented; the following list tries to give a general idea of the possible fields of study.

- Smart Home
- Robotic
- Aviation
- Automation of industries
- Biomedicine
- Monitoring of factories
- Smart Grids and Smart City's
- Military
- Sports
- Fraud detection
- Transportation systems

- Climatology
- Firefighters (heat sensors)

In the picture 8 it's exposed a little the idea set out above. The IoT together with the different boards designed for it, will allow us to control different elements of our life remotely improving the efficiency and saving of resources.

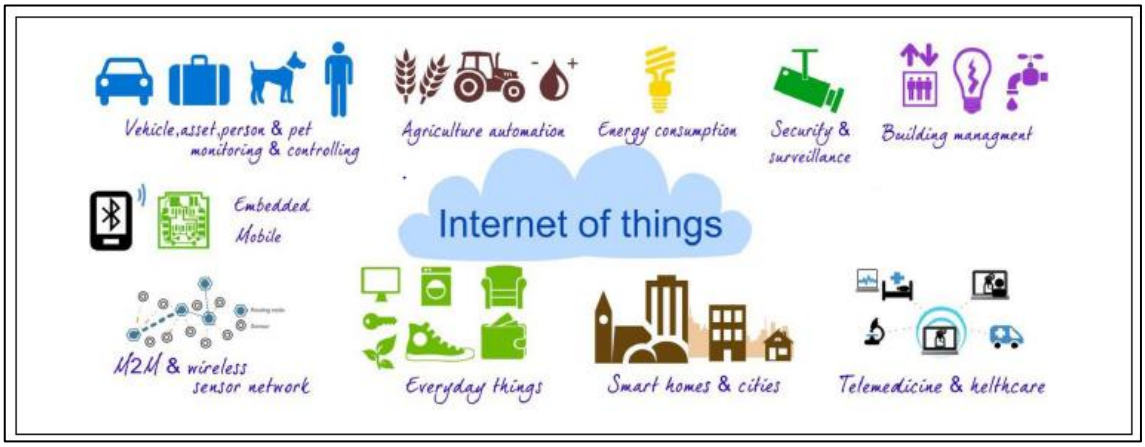


Figure 8. Uses cases for Internet of Things [3]

1.3.1 Future expectations

According to *Gartner Inc*, over 2020 there will be around 26 billion devices connected to the IoT [17], so all the necessary infrastructure has to be prepared to be able to services all these news systems.

Many experts say that IoT will radically change the way we live today [10]. For example, new smart devices will be able to save energy (either locally as a house or more generally as a city).

When integrating these new devices to the internet, a number of problems arise that must be considered, among the main ones we can highlight the use of IP addressing based on IPv4 where only $\mathbf{2^{32}}$ip addresses can be used, which means that IoT must works over IPv6 protocol that allow to work with $\mathbf{2^{128}}$ ip addresses.

Another disadvantage is the among of data that has to be transmitted and analyzed between the different devices interconnected in the IoT, so they cannot be treated in the traditional was because this will generate a would greatly delay in the communications, so we must focused on reach a solution where this data can be treated with big-data tools into a distributed system.

In addition to the previously mentioned, there is the most important aspect within the IoT that is security, it was defined by Professor Alberto Tejero Lopez [10] as "in its more technical aspect, it can be defined as those activities focused on protecting a particular device or service, as well as everything that interacts with and exchanges with other devices or services, be it informational, data, signals, etc. Based on the above definition, the security of the IoT could be defined as those activities aimed at the protection of objects and their communication or interact with other objects".

Faced with this new technologies scenario, should expand the studies and research about it, one of them being our proposal, where the performance of the Intel Galileo Gen 2 is evaluated against well-known algorithms.

1.4 Objectives of the project

The main objective of the project is to develop a grid computing on the intel Galileo Gen 2 board, in order to study the behavior of certain algorithms, These algorithms (with different degree of parallelism and requirement of systems resources) were programmed sequential and in parallel to study their execution time.

General Objectives

- Search information about Grid computing, openMP and MPI.
- Study of different algorithms.
- Search information about scientific papers related to our proposal.

Specific Objectives

- Define and explain the terms Grid computing and MPI.
- Study the basic characteristics of embedded systems.
- Implementation of the algorithms studied in sequential and parallel.
- Define the performance variables in parallel programing systems.
- Study and evaluated different algorithms with different degree of parallelism.
- Analyze and compare results from scientific publications to the results obtained by our proposal.

1.5 Working plan

Steps	October 2016	November 2016	December 2016	January 2017	February 2017	March 2017	April 2017	May 2017	June 2017
Research and theoretical research of grid computing and MPI									
Study of embedded systems and parallel programming									
Writing the theoretical framework									
Development and implementation of experimental and results									
Analyze the results obtained taking into account the variable raised									
Compare the results with the theoretical framework									
Drawing conclusions									

Chart 2. Working plan

Capítulo 2. Implementación de algoritmos

Para el desarrollo del proyecto se han programado y ejecutado sobre nuestro grid diferentes algoritmos (tanto en secuencial como en paralelo) para evaluar las diferentes características hardware que existen en las placas Intel Galileo Gen 2.

Los algoritmos que se han implementado tienen diferentes grados de dificultad y paralelismo, con la finalidad de estudiar la variable del tiempo que consumen cada uno de ellos.

A continuación, se explica brevemente la funcionalidad de cada algoritmo y el razonamiento por los cuales fueron escogidos, además de todas las herramientas utilizadas para lograr que el proyecto pudiera desarrollarse satisfactoriamente.

Elección de algoritmos

Para la selección de cada algoritmo se buscaron diferentes características con la finalidad de evaluar los diferentes aspectos hardware (CPU, Memoria RAM, memoria cache, etc.) de las placas Intel Galileo Gen 2.

El primer algoritmo escogido fue la Serie $\boldsymbol{x^2}$ ya que se necesitaba de un algoritmo sencillo para comenzar a comprender el funcionamiento de los programas ejecutados en paralelo (para esto se utilizó MPI ya que es un estándar sencillo que permite trabajar con programación paralela) y a su vez, un problema que no consumiera demasiados recursos de CPU y memoria cache. Esto permitió comprender cómo se comportan los sistemas distribuidos y los programas ejecutados en red para posteriormente utilizar estos conocimientos obtenidos en los siguientes algoritmos.

El segundo algoritmo fue el Quicksort, un algoritmo muy conocido y que ayudó a entender el comportamiento del sistema al trabajar con grandes cantidades de memoria. Este algoritmo requiere mucho más recursos que la Serie $\boldsymbol{x^2}$, ya que hace uso de una mayor cantidad de memoria RAM, cache y CPU. Otra de las características por las que se escogió trabajar con el Quicksort, es que al tener comunicar grandes

cantidades de datos, no se conocía el impacto real que el retardo de las comunicaciones podría tener. Esta fue una de las razones principales de la elección de este algoritmo

Como tercer algoritmo se escogió la multiplicación de matrices ya que éste combina la necesidad de utilizar grandes cantidades de memoria RAM junto a la gran cantidad de cálculos a realizar por parte de la CPU. Este algoritmo se podría considerar el más complejo y costoso (a nivel de recursos) utilizado en nuestra propuesta, por lo que se utilizó como referencia el trabajo realizado por ***Sherihan Abu ElEnin y Mohamed Abu [18]*** donde se analiza dicho algoritmo.

Para finalizar, se procedió con la implementación de un cuarto algoritmo que permitiera aproximar el número π (pi) mediante el método de Montecarlo. Este algoritmo no necesita de grandes capacidades hardware ni tampoco es un algoritmo complejo, pero nos pareció interesante incluirlo en nuestra propuesta ya qué aun siendo muy similar (en cuanto a requerimientos hardware se refiere) al algoritmo de Serie $\boldsymbol{x^2}$, se quería comprobar si al paralelizar el trabajo entre las diferentes placas se lograría obtener un número π más aproximado.

2.1 Cálculo de la Serie x2

Es un algoritmo sencillo de complejidad O(N), donde se calcula la suma de los cuadrados naturales hasta llegar a un determinado número N.

Su fórmula matemática puede verse a continuación:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Dicha igualdad se puede comprobar mediante inducción simple [21] la cual viene expuesta a continuación en el sub-capítulo 2.1.1.

2.1.1 Demostración matemática por inducción simple de la Serie x^2

El primer paso a comprobar es si el caso base es correcto, para esto se debe sustituir el valor de "n" por cero en ambas partes de la igualdad y comprobar que son idénticos, en nuestro caso base se cumple y se puede demostrar de la siguiente forma:

- Caso base (n = 0): $\sum_{i=1}^{0} x^2 = 0 = \frac{0*(0+1)*((2*0)+1)}{6}$

Una vez comprobado el caso base, se debe de cumplir la siguiente hipótesis de inducción (para cualquier número k ≥ 0, siendo k cualquier número perteneciente al conjunto de los números naturales), dicha hipótesis se puede apreciar a continuación:

- Hipótesis de inducción (HI): suponemos que:

$$\sum_{i=1}^{k} x^2 = \frac{k*(k+1)*(2k+1)}{6}$$

- A continuación se expone el paso inductivo simple donde se intenta comprobar si la HI se sigue cumpliendo para un numero k + 1:

$$\sum_{i=1}^{k+1} x^2 = (k+1)^2 + \sum_{i=1}^{k} x^2$$

$$= (k+1)^2 + \frac{k*(k+1)*(2l+1)}{6}$$

$$= (k^2 + 2k + 1) + \frac{2k^3 + 3k^2 + k}{6}$$

$$= \frac{6k^2 + 12k + 6 + 2k^3 + 3k^2 + k}{6}$$

$$= \frac{2k^3 + 9k^2 + 13k + 6}{6}$$

Por otra parte tenemos lo siguiente:

$$\frac{(k+1)*\big((k+1)+1\big)*(2*(k+1)+1)}{6} = \frac{(k+1)*(k+2)*(2k+3)}{6}$$

$$= \frac{\big(k^2+3k+2\big)*(2k+3)}{6}$$

$$= \frac{2k^3 + 6k^2 + 4k + 3k^2 + 9k + 6}{6}$$

$$= \frac{2k^3 + 9k^2 + 13k + 6}{6}$$

Habiendo obtenido el mismo resultado final en ambas cadenas de igualdades, podemos concluir que, como se quería demostrar:

$$\sum_{i=1}^{k+1} x^2 = \frac{(k+1) * \big((k+1)+1\big) * (2 * (k+1)+1)}{6}$$

La explicación de cómo se implementó dicho algoritmo se hará de forma más específica en los sub apartados 2.1.1 y 2.1.2.

2.1.2 Cálculo de la Serie x^2 secuencial

Tal y como se puede intuir de la fórmula matemática del apartado 2.1, se implementó la serie con un único *for* de manera que se calcularan las sumas parciales hasta llegar al número final N.

A continuación podemos ver el código utilizado para realizar la suma:

```
For (int i = 1; i <= N; i++){
        sumaTotal += sumaTotal + i*i;
}
```

Una vez terminado el bucle for, la variable sumaTotal contendrá el resultado final de la serie N^2. Los detalles de este algoritmo se pueden apreciar en el anexo A.

2.1.3 Cálculo de la Serie x2 paralelo

El diseño del algoritmo fue implementado de la siguiente forma:

- Se tiene un número K (siendo K >= 2) de placas donde se va a ejecutar el programa.
- La placa principal, antes de comenzar con la etapa de cálculo, obtiene el número N (siendo N el límite superior de la suma cuadrática) para posteriormente comunicárselo al resto de placas. Un ejemplo de cómo transferir datos a las diferentes placas puede ser la siguiente función de MPI:

```
MPI_Bcast( void* data, int count, MPI_Datatype datatype,
int root,  MPI_Comm communicator);
```

Para explicar mejor dicha función, a continuación se expone un extracto del código utilizado para transferir datos en el algoritmo de serie $\boldsymbol{x^2}$:

```
Double N = 0;
Int master = 0;

MPI_Init(&argc, &argv);
N = atoi(argv[1]);

MPI_Bcast(N, 1,MPI_DOUBLE,Master,MPI_COMM_WORLD);
```

La variable N (como bien se explicó anteriormente) contiene el límite superior que todas las placas deben conocer, el segundo argumento "1" representa la cantidad de datos a transferir (si se pusiera un valor superior se le estaría indicando al resto de placas que se intenta transferir un array de datos), el tercer argumento representa el tipo de datos a transferir, en nuestro caso es MPI_DOUBLE ya que la variable es de tipo double, el cuarto argumento denominado "Master" representa él emisor de los datos (la placa principal siempre tiene el valor "0"); Para finalizar, el último argumento representa la forma en que se envían los datos, en este método seleccionamos el modo broadcast ya que es necesario que todas las placas conozcan el límite superior.

Cabe destacar que el dominio de cálculo (todos los números naturales comprendidos entre 1 y N) se distribuye de forma equitativa entre las distintas placas.

- Una vez transmitido el número N a todas las placas, éstas comienzan con la fase de cálculo de las sumas parciales (incluyendo la placa principal). Un ejemplo de cómo se realizaría este cálculo con 4 placas se muestra en la ilustración 9 donde el número N sería 16:

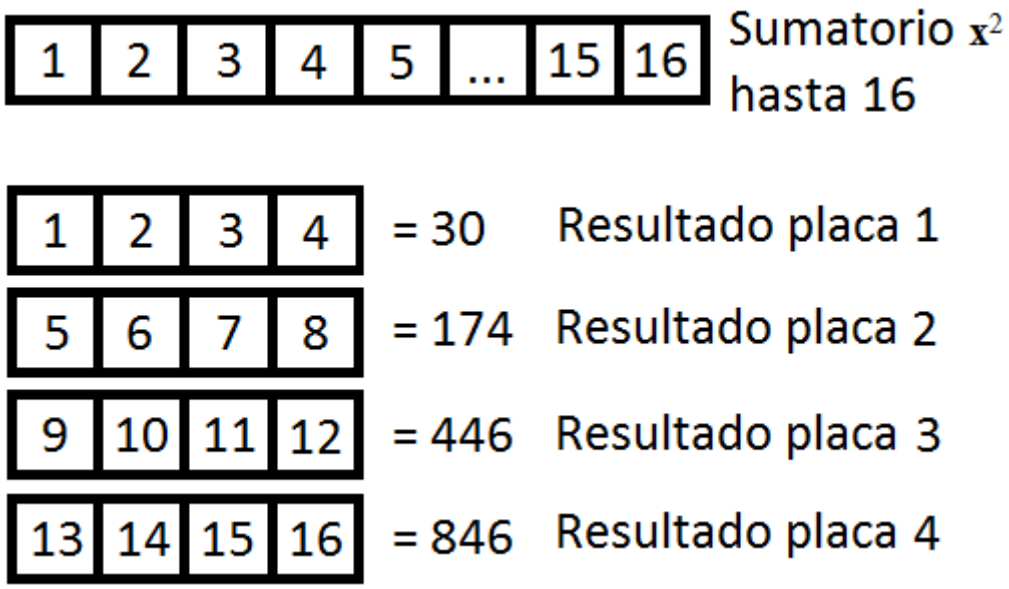


Ilustración 9 explicación algoritmo Serie x^2 en paralelo.

En la explicación anterior se expuso un ejemplo concreto de cuando el límite superior N es múltiplo al total del número de placas, a continuación se expone la fórmula general para calcular la agregación para cualquier límite superior N y para cualquier número de placas K (para un K incluido dentro del conjunto de los números naturales y superior o igual a dos):

1. Primero se debe de definir el caso base, el cual sería la suma parcial para la placa principal con id (identificador) igual a "0"

$$\mathbf{0} \leq \boldsymbol{i} < \left(\frac{\boldsymbol{N}}{\boldsymbol{K}}\right)$$

La variable "i" es un numero comprendido en el conjunto de los números naturales.

2. Si la placa no es la principal, quiere decir que su id es superior a cero, por lo que sus límites se calculan de la siguiente forma:

$$\boldsymbol{id} * \left(\frac{\boldsymbol{N}}{\boldsymbol{K}}\right) \leq \boldsymbol{i} < (\mathbf{2} * \boldsymbol{id}) * \left(\frac{\boldsymbol{N}}{\boldsymbol{K}}\right)$$

- La placa principal, una vez obtenido todas las sumas parciales, está realiza una última operación de cálculo que consiste en sumar todos los resultados. Un ejemplo de cómo se reciben datos en MPI es el siguiente:

```
MPI_Recv( void* data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status* status)
```

A continuación se pone un extracto del código para entender mejor el funcionamiento del método MPI_Recv:

```
Int sumaParcial = 0;

MPI_Recv(sumaParcial, 1, MPI_DOUBLE, slave-1, Master, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

La única variación que hay con los argumentos de esta función y los de MPI_Bcast es que el argumento llamado "slave-1" representa el identificador de la placa que envía el resultado parcial obtenido a la placa principal, el último argumento comunica a la placa principal si ha tenido algún problema al calcular su parte del problema, este al no haber sido utilizado en nuestra propuesta se le pone como argumento MPI_STATUS_IGNORE que significa no evaluar el estado de las placas slaves.

- Para finalizar, la placa principal imprime el resultado final, en nuestro ejemplo anterior ésta imprimiría como resultado 1496. Los detalles de este algoritmo se pueden ver en los anexos B y C.

2.2 Algoritmo de ordenación Quicksort

Se trata de un Algoritmo de ordenación creado por el científico británico ***Charles Antony Richard Hoare*** basado en la técnica de divide y vencerás que permite ordenar N elementos en un tiempo proporcional a O(N·log N). Fue desarrollado mientras su creador se encontraba visitando la universidad de ***Moscow State*** trabajando para el proyecto ***Machine Translation*** en el laboratorio ***National Physical Laboratory***. Trabajando en la universidad de ***Moscow*** tendía a utilizar mucho el diccionario para poder entender lo que decían sus compañeros; por ello, como parte del proceso de traducción, el científico ordeno alfabéticamente las palabras que deseaba traducir y posteriormente las buscaba en su diccionario de inglés-ruso.

Este es solo un ejemplo de los múltiples problemas a los que se enfrentan los diseñadores de software, ya sea que se esté intentando ordenar una base de datos de direcciones Web, un listado de clientes o cualquier problema en el que se necesite de una ordenación previa de los datos de entrada.

A continuación en el apartado 2.2.1 se explicara cómo funciona el algoritmo quicksort de manera secuencial.

2.2.1 Algoritmo de ordenación Quicksort secuencial.

En el presente sub-capítulo se ilustrará cómo funciona el algoritmo de quicksort diseñado por ***Richard Hoare***, aunque el algoritmo fue diseñado originalmente para ordenar palabras (cadena de caracteres) en nuestra implementación se ordenaran variables de tipo entero (int).

2.2.1.1 Funcionamiento algoritmo Quicksort

El Quicksort primero divide un array (de longitud n) en dos sub-arrays de longitud $\frac{n}{2}$ donde en el primer array se incluyen los elementos más pequeños y en el otro los más grandes.

A continuación, se muestran los pasos básicos del funcionamiento del algoritmo:

1. Se utiliza un elemento del array al que se le denominará pivote. Para la elección de dicho elemento se suele utilizar una función de selección para intentar aproximar lo más posible al elemento medio (dicha función se le denomina pivote en nuestra implementación). La elección del dicho elemento se realiza de la siguiente manera:

```
Int pivote (int a[], int l, int r){
        Int pivot, i, j, l;
        pivot[l] = a[l];
        i = l; j = r+1;
        while (1){
                do ++i; while( (a[i] <= pivot) && (i <= r) );
                do --j; while(a[j] > pivot);
                if( i >= j ) break;
                t = a[i]; a[i] = a[j]; a[j] = t;
        }
        t = a[l]; a[l] = a[j]; a[j] = t;
        return j;
}
```

La variable i se inicializa con cero y se va incrementando con cada iteración del bucle while siempre y cuando el elemento a[i] sea inferior al elemento pivote, esta condición se evalúa siempre y cuando la variable i sea inferior a la variable r (siendo r la longitud del array).

La variable j se inicializa con la longitud del array y esta va decreciendo siempre y cuando el elemento a[j] sea mayor al elemento pivote.

Como última condición se evalúa que la variable i no sea superior al elemento j, ya que si sucediera ya se habrían interceptado los índices.

2. Se realiza una ordenación parcial sobre el pivote donde todos los elementos más pequeños que ese pivote estarán a la izquierda en el sub array 1 y los más grandes a la derecha en el sub array 2. Después de esta partición, el pivote está en su posición final, a la cual se le denomina partition operation.
3. Ahora recursivamente se aplican los pasos descritos anteriormente sobre los sub arrays 1 y 2 de elementos. El elemento pivote se vuelve a calcular sobre dichos sub arrays y el proceso continua hasta llegar a los casos base (sub arrays con un único elemento o sub arrays sin elementos).

En la ilustración 10 se explica de forma gráfica la funcionalidad del Quicksort.

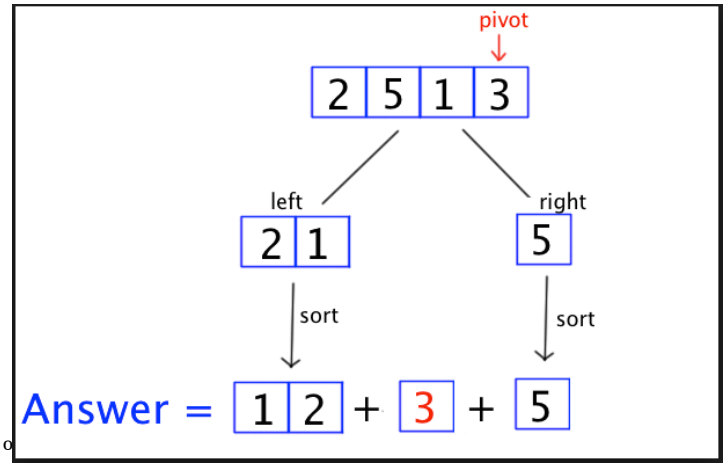


Ilustración 10 Ejemplo de ejecución del algoritmo Quicksort

La eficiencia de dicho algoritmo depende mucho de la posición en la que termine el elemento pivote. En el mejor de los casos puede quedar a la mitad de la lista dando como resultado un división casi homogénea entre los 2 sub arrays, se dice que en este caso el orden de complejidad promedio del algoritmo es de $O(n \cdot \log n)$. Sin embargo, en el peor de los casos, cuando el elemento pivote termina en uno de los extremos de la lista, el orden de complejidad del algoritmo se incrementa hasta $O(n^2)$. Esto habitualmente sucede cuando la lista ya se encuentra ordenada o semi ordenada. Un ejemplo de esta problemática se puede ver en la ilustración 11. El caso base de la recursión es un array sin elementos o con un solo elemento, los cuales nunca necesitan ser ordenados.

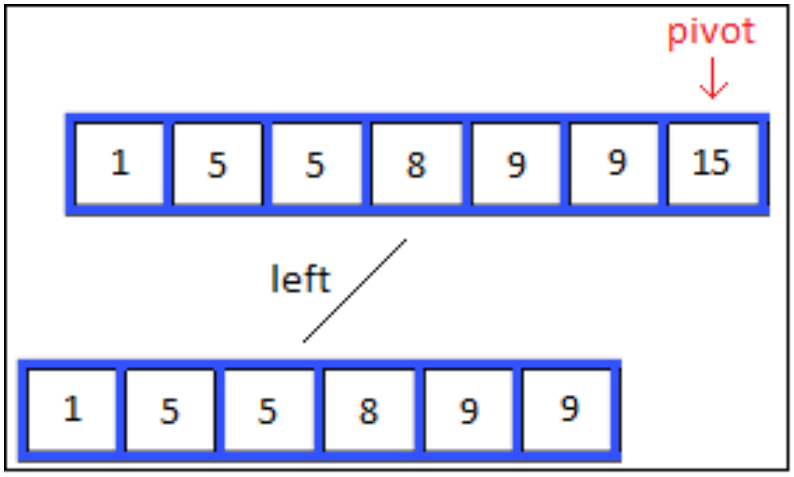


Ilustración 11 Problemas algoritmo Quicksort con arrays ordenados.

Adicionalmente el algoritmo presenta otro problema, y es que cuando el programa trabaja con elementos repetidos éste muestra un rendimiento bastante bajo debido a la gran cantidad de elementos similares entre sí. En cada recursión del algoritmo éste genera algún sub array vacío mientras que todos los elementos (menos el elemento pivote) se encuentran en el otro sub array. Una de las soluciones creadas para resolver este problema es generar 3 grupos relacionados al elemento pivote; los elementos inferiores al elemento pivote y no repetidos, los elementos que son iguales, y los mayores al pivote, éste último en nuestro ejemplo sería un sub array sin elementos; En la ilustración 12 mostramos un ejemplo de cómo resolver esta problemática.

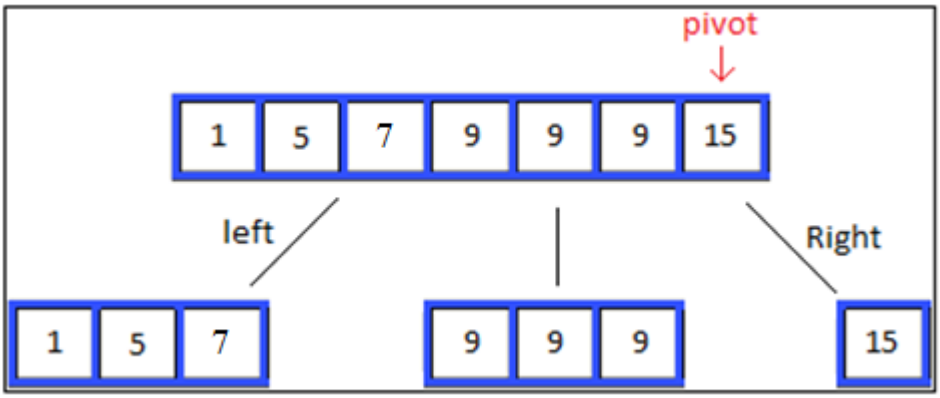


Ilustración 12 solución al problema Quicksort con elementos repetidos.

De esta manera el algoritmo solo tendría que realizar la ordenación del sub array de la izquierda que tiene como elementos el 1, 5 y 7. Al regresarlo ya ordenado el algoritmo vuelve a introducir los elementos repetidos más el elemento pivote devolviendo el resultado final.

El código asociado a la implementación de nuestro algoritmo se puede ver en el anexo D. En la siguiente sub sección 2.3.2 se analizará cómo funciona el algoritmo quicksort trabajando de forma paralela.

2.2.2 Algoritmo de ordenación Quicksort paralelo

En las secciones anteriores se explicó cómo funciona el algoritmo quicksort, en este sub apartado analizaremos como dividir el trabajo entre las diferentes placas.

Para comenzar la placa principal genera P sub arrays (siendo P igual al número de placas trabajando para resolver el problema) donde cada una de ellas estarán almacenados un conjunto de los datos a ordenar; la agregación de todos los sub arrays generará el array final de tamaño K (siendo K el tamaño del problema a solucionar). Para demostrar gráficamente esta idea se ilustra a continuación cómo se ordenaría un array de 20 elementos con 4 placas.

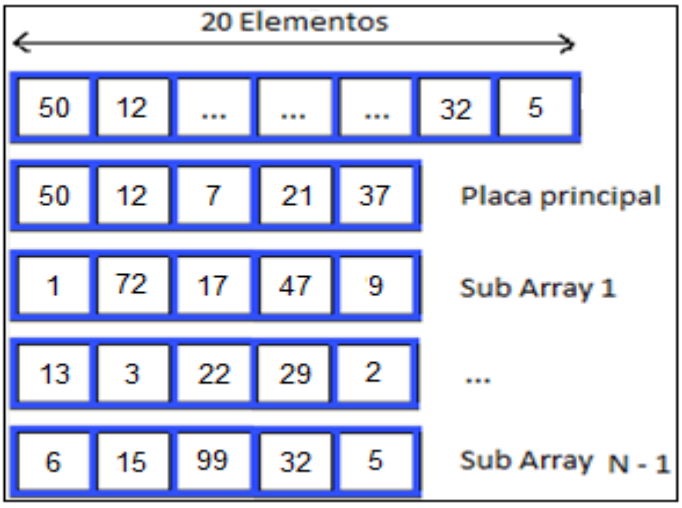


Ilustración 13 explicación grafica de nuestra implementación del algoritmo Quicksort en paralelo.

En el ejemplo de la ilustración 13 se pudo observar como la placa principal genera 4 sub arrays (en el ejemplo se simula un problema para 4 placas) de longitud 5. La agregación de estos 4 sub arrays generaría el array original de 20 elementos. Una vez generado estos arrays la placa principal se encarga de transferir los N-1 sub arrays al resto de placas. Un ejemplo de como lo haría se puede ver en la ilustración 14.

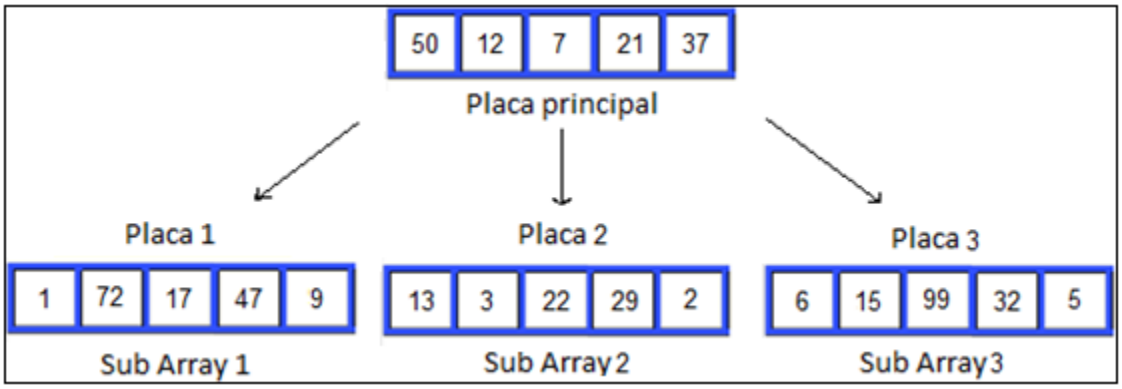


Ilustración 14 división de trabajo en el algoritmo Quicksort.

Una vez transmitidos todos los sub arrays, cada placa comienza con la fase de ordenación de forma independiente, de forma que, al finalizar las ordenaciones parciales las diferentes placas transmiten sus resultados a la placa principal. En la ilustración 15

se representa de forma gráfica los resultados obtenidos del ejemplo expuesto por la ilustración 14.

En lugar de generar un array de tamaño K, se consideró más eficiente un algoritmo que generará $\frac{K}{P}$ arrays (siendo P igual al número de placas y K el número total de elementos) para volcar los resultados obtenidos por las diferentes placas, una vez la placa principal obtuviera todos los resultado, esta realiza una última operación en la que consistiría en imprimir los resultados, para esto se implementó una función denominada merge que realiza una búsqueda entre los elementos más pequeños de los $\frac{K}{N}$ arrays generados, si alguno de los elementos estuviera repetido él algoritmo imprimiría el primero de los elementos evaluados, si uno de los sub arrays estuviera vacío (sin más elementos que evaluar) la función se centraría en buscar el elemento más pequeño en los sub arrays restantes. Un ejemplo de cómo funciona dicha función se puede observar en la ilustración 15.

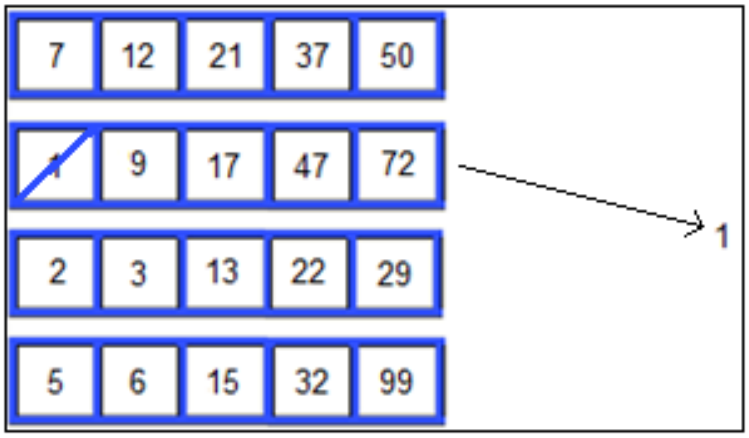


Ilustración 15 Imprimir resultado de ordenación con MPI.

El proceso solo termina cuando ningún sub array tenga elementos a imprimir, para aclarar un poco más esta función realizaremos una interacción más.

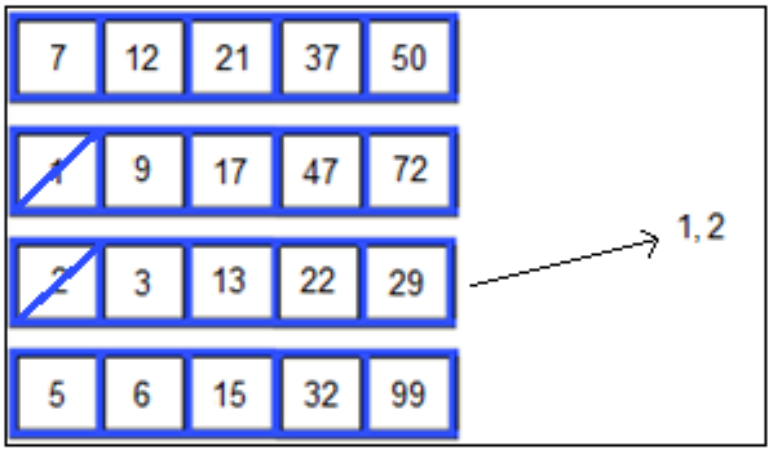


Ilustración 16 Segunda iteración función merge algoritmo Quicksort MPI.

El proceso continua ejecutándose mientras existan elementos en los sub arrays. Una vez agotados todos los elementos el programa termina. En la ilustración 17 mostramos el resultado de realizar las 20 iteraciones del ejemplo expuesto.

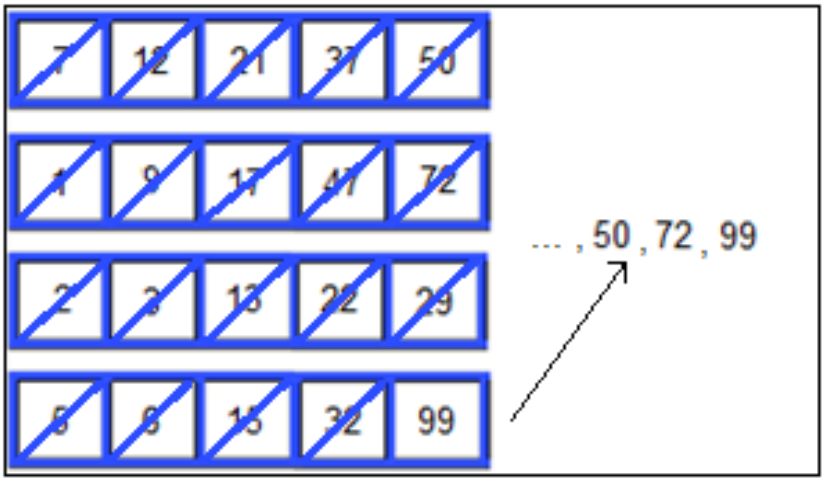


Ilustración 17 Ultima iteración función merge algoritmo Quicksort MPI

En los anexos E y F se puede ver el código para la ejecución de dicho algoritmo.

2.3 Algoritmo Multiplicación de matrices

En matemática se denomina producto matricial [22] a una operación binaria de operaciones donde a partir de 2 matrices A (n x m) y B (m x p) (siendo n, m y p las dimensiones de las matrices) produce una nueva matriz AB de dimensiones (n x p). Se utiliza principalmente en las ecuaciones lineales, teniendo un gran número de aplicaciones en ramas como las matemáticas, ingeniería y física. El desarrollo matemático se podrá entender mejor en las siguientes ilustraciones.

Supongamos que tenemos las siguientes matrices iniciales y se desea conocer el resultado de la multiplicación AB.

$$\mathbf{A} = \begin{pmatrix} A_{11} & \cdots & A_{1m} \\ A_{21} & \cdots & A_{2m} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B_{11} & \cdots & B_{1p} \\ B_{21} & \cdots & B_{2p} \\ \vdots & \ddots & \vdots \\ B_{m1} & \cdots & B_{mp} \end{pmatrix}$$

Ilustración 18 Matrices iniciales [22]

El resultado final de la matriz AB seria el siguiente:

$$\mathbf{AB} = \begin{pmatrix} (\mathbf{AB})_{11} & \cdots & (\mathbf{AB})_{1p} \\ (\mathbf{AB})_{21} & \cdots & (\mathbf{AB})_{2p} \\ \vdots & \ddots & \vdots \\ (\mathbf{AB})_{n1} & \cdots & (\mathbf{AB})_{np} \end{pmatrix}$$

Ilustración 19.Resultado de multiplicar la matriz A y B [22].

A continuación se explicará más al detalle cómo se realizan los cálculos de cada uno de los elementos de la nueva matriz AB.

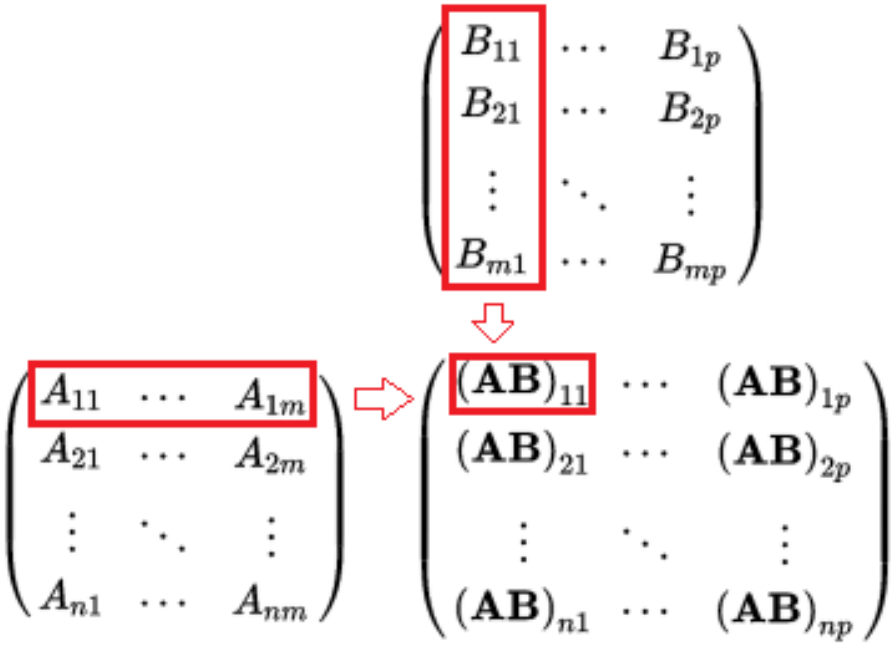


Ilustración 20 . Multiplicación de matrices

Cada sub elemento de la nueva matriz se puede calcular con la siguiente fórmula matemática:

$$(\mathbf{AB})_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$

2.3.1 Multiplicación de matrices secuencial

Como se pudo ver en la ilustración 20, cada nuevo elemento de la nueva matriz se calcula realizando la fórmula matemática expuesta anteriormente. Es un proceso bastante laborioso que requiere una gran cantidad de cálculos por lo que la mejor forma de realizarlo es a través de un ordenador. A continuación mostramos el código en C++ para realizar dicho cálculo.

```
for(int i=0; i<n; ++i)
        for(int j=0; j<n; ++j)
                for(int k=0; k<m; ++z)
                        AB[i][j] += A[i][k] * B[k][j];
```

Para simplificar los cálculos y poder compararlos con otros proyectos de investigación, se procedió a utilizar únicamente matrices cuadradas, esto quiere decir que los índices de todas las matrices de nuestros experimentos son (n x n). Aunque sólo se hayan realizado cálculos para matrices cuadradas esto no quiere decir que el programa no acepte matrices con diferentes dimensiones (siempre y cuando se mantengan las propiedades básicas de la multiplicación de matrices). Para observar el código de dicho programa ir al anexo G.

En la siguiente sub sección se explicará brevemente como se realizó dicho cálculo trabajando con múltiples placas.

2.3.2 Multiplicación de matrices en paralelo

Como ya se estudió en la sección anterior, el procedimiento para calcular el valor de ABij es conocida, por lo que ahora procederemos a explicar cómo repartir los cálculos entre las diferentes placas.

Al ser siempre matrices cuadradas (nxn), esto nos permite simplificar mucho el trabajo de repartición, ya que lo que se hace es dividir el número de filas N entre K (siendo K el número de placas) permitiendo que cada placa sólo calcule una sección de la matriz AB. En la siguiente ilustración explicamos de forma gráfica esta idea suponiendo que lo que se intenta calcular es la matriz AB con 2 placas.

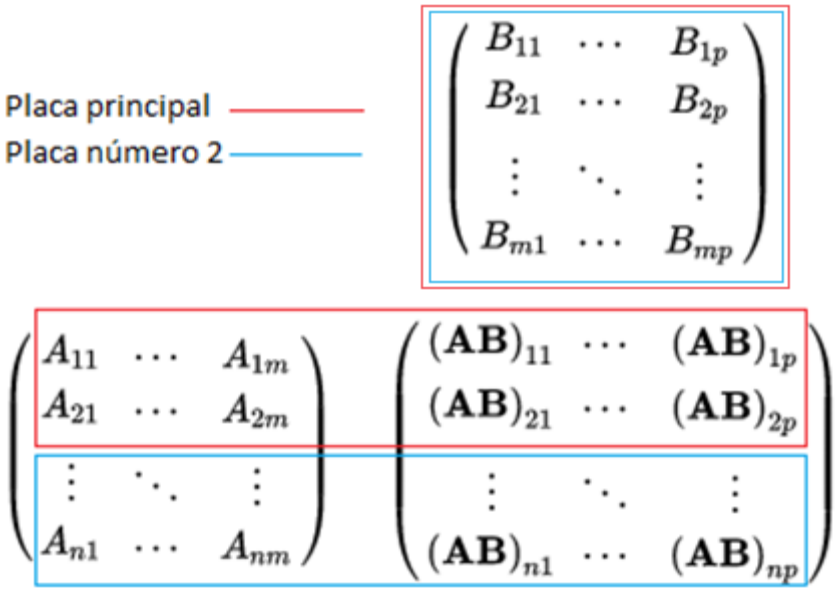


Ilustración 21 . Dividir filas entre las diferentes placas con MPI.

Como se puede observar en la ilustración 21, lo que hacemos es dividir el rango de i con la finalidad de que cada placa tenga una parte de la solución final. Para explicar mejor esta idea la ilustraremos con el código para su cálculo con 2 placas

```
If (my_id == 0){
        for(int i=0; i<n/2; ++i)
                for(int j=0; j<n; ++j)
                        for(int k=0; k<m; ++z)
                                AB[i][j] += A[i][k] * B[k][j];
}else{
        for(int i=n/2; i<n; ++i)
                for(int j=0; j<n; ++j)
                        for(int k=0; k<m; ++z)
                                AB[i][j] += A[i][k] * B[k][j];
}
```

Una vez que cada placa termine su fase de cálculo lo único que faltaría es copiar estos resultados a la matriz final de la placa principal para posteriormente imprimirlos por pantalla. Para entender mejor el funcionamiento de dicho algoritmo consultar al anexo H.

Con esto se da por finalizado la explicación del algoritmo de multiplicación de matrices, en la sección posterior se explicará el ultimo algoritmo desarrollado en nuestra propuesta.

2.4 Algoritmo para calcular π mediante el método de Montecarlo

Es el único algoritmo que no se implementó con la intención de mejorar los tiempos de ejecución, la finalidad de este algoritmo es comprobar si al ampliar el número de placas el valor estimado de $\boldsymbol{\pi}$ es cada vez más próximo a su valor real. Este método matemático se fundamenta en utilizar funciones aleatorias las cuales permiten aproximar el valor de π por un método no determinista. A continuación en los sub apartados 2.4.1 y 2.4.2 se explicara cómo funciona dicho método y como se implementó.

2.4.1 Explicación método de Montecarlo

Es un método matemático que permite estimar de forma no determinista el valor de π. La idea en esencia es la de utilizar el azar para resolver problemas que en un principio se pueden considerar deterministas. Este método genérico es utilizado principalmente para resolver problemas matemáticos y físicos siendo estos clasificados en 3 grandes grupos: optimización, análisis numérico y problemas de distribución de probabilidades.

Su metodología puede parametrizarse de la siguiente manera:

1. Definir el dominio de los datos de entrada.
2. Definir una función que genere datos aleatorios sobre el dominio definido en el paso 1.

3. Realizar un cálculo determinista sobre los datos obtenidos.
4. Agregar lo obtenido al resultado parcial.

Una vez explicado cómo funciona el método de Montecarlo, se procederá a explicar cómo se calculó el valor de π.

Lo primero es definir un área que aproxime lo más posible el número π, es por esta razón que se optó por elegir el área del circulo ya que si dividimos la longitud de la circunferencia entre su diámetro da como resultado el número π. Una vez definida esté área, se debe proceder a buscar otro área que encierre el área del círculo, siendo la más adecuada para simplificar los cálculos el área del cuadrado. Un ejemplo de cómo cubrir dichas áreas se puede apreciar en la ilustración 22.

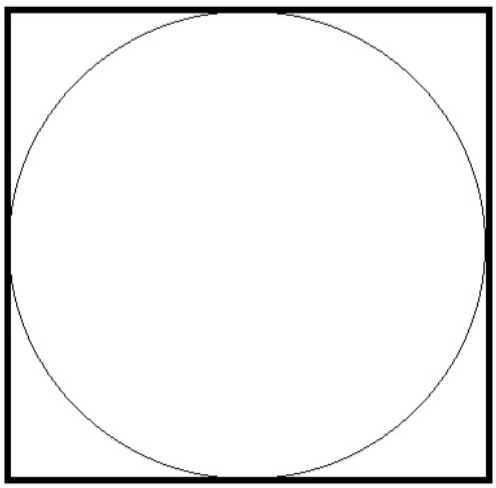


Ilustración 22 Definición de áreas en algoritmo de Montecarlo.

Con la finalidad de no desperdiciar recursos en las placas Intel galileo gen 2 y simplificar los cálculos, se procedió a dividir en 4 partes el área expuesta en la ilustración 22, de esta manera el valor que obtenemos al final de la ejecución del algoritmo es $\frac{\pi}{4}$. Esta idea se puede apreciar mejor en la ilustración 23.

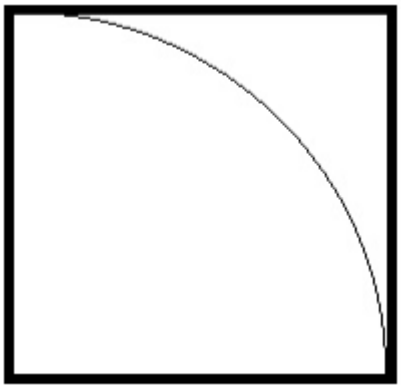


Ilustración 23 Área reducida algoritmo Montecarlo.

Una vez definido el área de la ilustración 23, es necesario conocer el área total del cuadrado siendo esta expresada con la siguiente formula:

$$\boldsymbol{Area_{Cuadrado} = l^2}$$

Ya habiendo definido el área del cuadrado, es necesario definir el área del semicírculo encerrado en el área del cuadrado, como la intención es simplificar los cálculos, el área del círculo debe de ser dividida en cuatro partes, quedando como resultado la siguiente fórmula matemática:

$$\boldsymbol{Area_{Círculo} = \frac{\pi r^2}{4}}$$

Para finalizar, es necesario conocer la diferencia entre las áreas previamente mencionadas, a continuación se formula el ratio entre el área del círculo y el área del cuadrado (como la longitud del lado del cuadrado es igual a la longitud del radio de la circunferencia, se anulan dando como resultado la siguiente fórmula matemática):

$$\frac{\boldsymbol{Área_{Circulo}}}{\boldsymbol{Área_{Cuadrado}}} = \boldsymbol{\frac{\pi r^2}{4l^2} = \frac{\pi}{4}}$$

Una vez terminado de definir el dominio de los datos de entrada, con esto se termina el punto 1 de la metodología descrita en el capítulo 2.4.1 (definir el dominio de cálculo), ahora se procederá a definir una función aleatoria que se encargue de generar puntos aleatorios dentro de dicha área, esta función junto a los programas completos se podrán apreciar en los anexos I y J.

Para finalizar con la ejecución del programa, lo que se debe de ejecutar es la función definida anteriormente un número de iteraciones lo suficientemente elevado con la finalidad de calcular la relación entre los puntos que cayeron dentro del área del semicírculo y los puntos totales generados. La funciona matemática que explica mejor esta idea es la siguiente:

$$\frac{\boldsymbol{Puntos_{Dentro\ del\ circulo}}}{\boldsymbol{Puntos\ totales\ generados}} \approx \frac{\boldsymbol{\pi}}{\boldsymbol{4}}$$

A continuación se explicará la eficiencia de este algoritmo frente a otros métodos de cálculo.

Este método es eficiente siempre y cuando se cumplan estas 2 condiciones:

1. Se genere números aleatorios de forma homogénea para intentar cubrir todo el espectro de trabajo. De esta forma, habrá más probabilidad de que la relación entre el total de puntos lanzados y los puntos dentro de nuestro objeto sea similar a la relación real entre el área de nuestro campo de pruebas y el objeto circunscrito que deseamos medir.
2. Que se ejecute el programa con un número elevado de iteraciones ya que si no el resultado obtenido puede no tener relación con el valor real que se desea calcular.

En la siguiente sub sección se explicará cómo se implementó el algoritmo de Montecarlo en programación paralela.

2.4.2 Calculo de π en programación paralela

Para realizar el cálculo de π en programación paralela se procedió a dividir el área del semicírculo en N zonas (siendo N igual al número de placas), un ejemplo de esta división se puede observar en la ilustración 24.

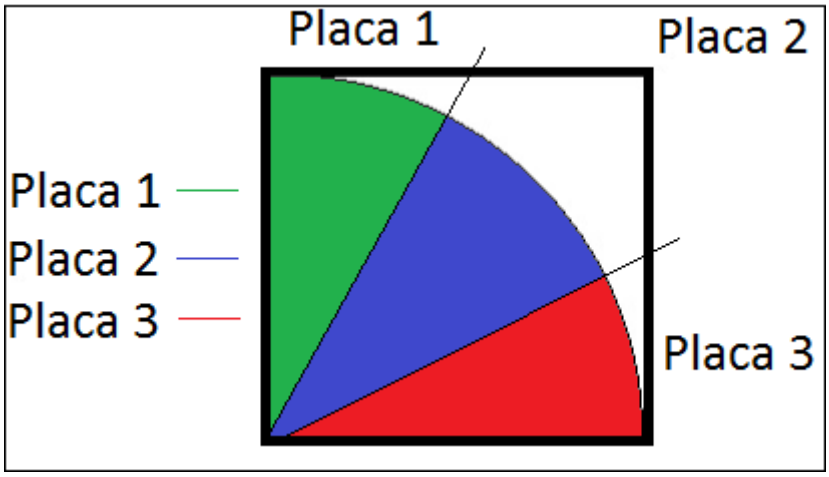


Ilustración 24 Repartición del área del circulo entre 3 placas.

De esta manera cada placa solo debe calcular una sub sección del área total acelerando el proceso de cálculo. Una vez dividido el área en N partes, la placa principal (en la imagen la placa 1) transfiere el número de iteraciones K al resto de placas (una iteración equivale al punto generado aleatoriamente por la función definida en el apartado 2.4.1 apartado 2), una vez que cada placa ha obtenido el número total de iteraciones, las placas proceden a calcular cuantas iteraciones debe de realizar, y esto viene definido por la siguiente formula:

$$\boldsymbol{N\acute{u}mero_{Iterciones}} = \frac{\boldsymbol{K}}{\boldsymbol{N}}$$

Cabe destacar que cada placa estimaría el valor $\boldsymbol{\pi}$ en $\frac{\boldsymbol{\pi}}{\boldsymbol{4N}}$

Una vez terminado el proceso de cálculo, la placa principal obtiene los diferentes resultados parciales y ésta (en la ilustración 24 sería la placa 1) realiza una última operación de cálculo sumando los resultados anteriormente obtenidos y

multiplicándolos por 4 para obtener el valor estimado de π para K iteraciones. Para entender mejor el funcionamiento de este algoritmo consultar los anexos I y J.

Capítulo 3. Entorno experimental

En el presente capítulo se describirán las características del entorno de desarrollo para nuestra propuesta, en particular se expondrán las características físicas de la placa Intel Galileo Gen 2, el sistema operativo utilizado (Debían 7) y las características principales de red (direccionamiento de red). Adicionalmente y para terminar con el capítulo, se explicarán una serie de utilidades implementadas a lo largo de la vida del proyecto que ayudaron a que el reparto de trabajo entre las diferentes placas, guardado de resultados y exposición de los resultados obtenidos resultasen más sencillos.

3.1 Placa Intel Galileo Gen 2

La placa Galileo Gen 2 [4] desarrollada por Intel® es la primera de una familia de Arduino* basada en la arquitectura Intel® y diseñada específicamente para creadores, estudiantes, educadores y entusiastas de la electrónica.

La placa Intel Galileo Gen 2 proporciona a los usuarios un entorno de desarrollo de software y hardware de código totalmente abierto. Además, complementa y amplía la línea de productos Arduino para ofrecer funciones informáticas más avanzadas a los usuarios ya familiarizados con las herramientas de prototipo Arduino.

3.1.1 Descripción general.

Entre sus características principales [4] podemos destacar las siguientes.

- Procesador de aplicaciones Intel® Quark™ SoC X1000, 32 bits, un núcleo, un hilo, compatible con la arquitectura de conjunto de instrucciones (ISA) del procesador Intel® Pentium®, que funciona a velocidades de hasta 400 MHz.
- Compatible con una amplia gama de interfaces de E/S estándar de la industria, como la ranura de tamaño completo mini-PCI Express*, el puerto Ethernet de 100 Mb, la ranura de microSD*, el puerto anfitrión USB y el puerto cliente USB.
- DDR3 de 256 MB, SRAM integrada de 512 KB, memoria NOR Flash de 8 MB; además, admite una tarjeta microSD de hasta 32 GB; Para terminar la placa también cuenta con una memoria cache de L1 de 16 KBytes.
- Compatibilidad de hardware y pines con una amplia gama de protectores Arduino Uno R3.
- Programable a través del entorno de desarrollo integrado (IDE) de Arduino, compatible con los sistemas operativos anfitriones Microsoft Windows*, Mac OS* y Linux.
- Compatible con la alimentación a través de Ethernet (PoE) de 12 V (es necesario instalar el módulo PoE).
- El sistema de regulación de energía se modificó para admitir fuentes de alimentación de 7 a 15 V.
- El Intel® IoT Developer Kit para Intel Galileo Gen 2 añade soporte para C, C++, Python y Node.js/Javascript para desarrollar aplicaciones del Internet de las cosas con sensores conectados.

Cabe destacar que la placa Galileo ha sido diseñada para funcionar con 3,3 V o 5 V. La tensión de funcionamiento principal de Galileo es de 3,3 V. Sin embargo, un puente en la placa permite una traducción de voltaje de 5 V en los pines de E/S. Esto permite que funcione a 5 V.

En la siguiente figura, se puede apreciar una imagen real de la placa Intel Galileo Gen 2.

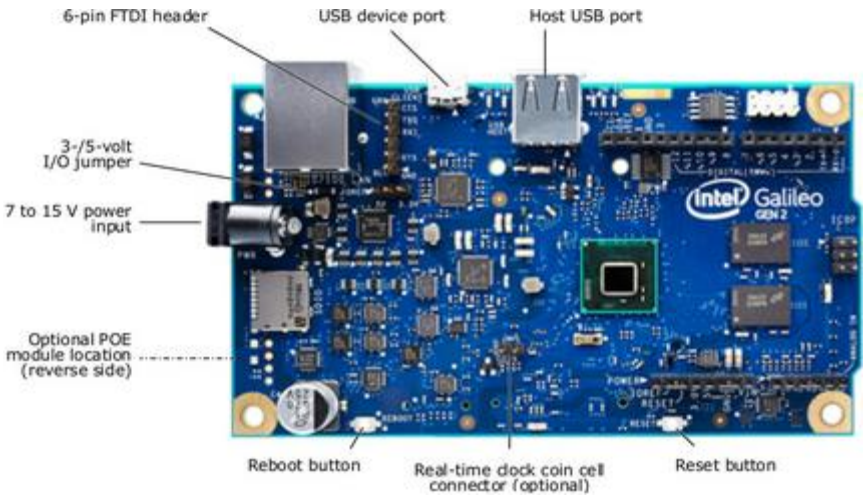


Ilustración 25. Placa Intel Galileo Gen 2 [4]

3.1.2 Características físicas

La placa Intel Galileo Gen 2 tiene las siguientes características físicas:

- 10 cm de longitud y 7 cm de ancho con los conectores USB, el conector UART, el conector Ethernet y el conector de corriente que es de mayor dimensión al anterior.
- Cuatro agujeros para tornillos permiten que la placa pueda fijarse a una superficie o carcasa.
- Botón de restablecimiento para restablecer el boceto y cualquier cubierta adjunta.
- Reloj de tiempo real (RTC) integrado con batería "tipo botón" de 3V opcional para operar entre los ciclos de encendido.
- Compatible con el estado de suspensión de CPU y con ACPI

3.1.3 Opciones de energía

La placa Intel Galileo Gen 2 da la opción de alimentarse a través de un adaptador AC-to-DC, insertando el adaptador de 2.1 mm en la placa. La salida recomendada de alimentación es de 5V.

A continuación, se dará un resumen eléctrico de la placa:

- Voltaje de entrada recomendado 5V
- Límite de voltaje de Entrada 5V
- 14 Pines digitales de Entrada/Salida
- La Salida total de los DC a las líneas de entrada/salida es de 80 mA
- Posibilidad de alimentarse a través de PoE (Power-over-Ethernet)

3.1.4 Características de Red

La placa Intel Galileo Gen 2 posee un conector Ethernet 10/100 permitiendo conexiones de hasta 100 Mbps. También da la posibilidad (como casi cualquier dispositivo pensado para trabajar en la IoT) de trabajar con PoE (Power Over Ethernet).

Para finalizar, se ilustra el diagrama de "interconexión" interno de la Intel Galileo Gen 2.

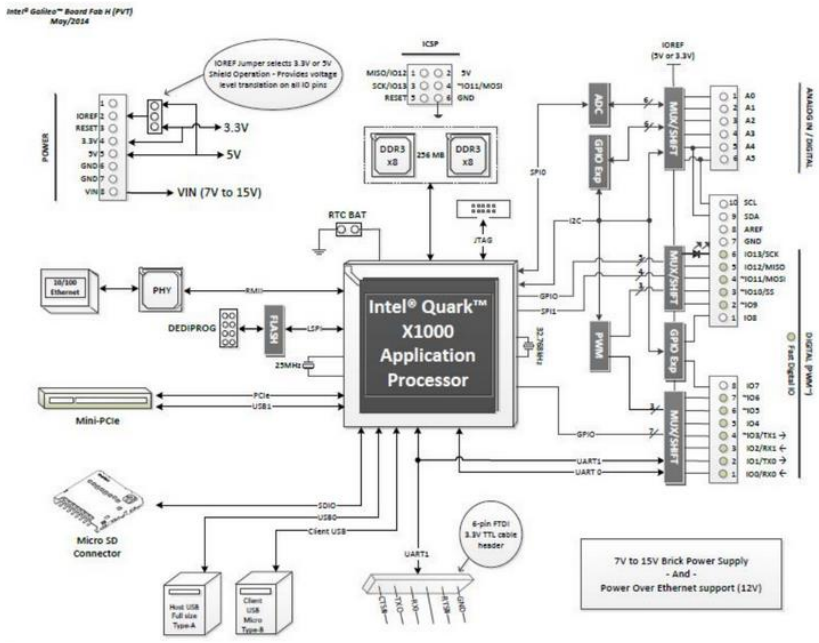


Ilustración 26. Intel galileo Datasheet [4]

3.2 Sistema operativo Debian 7

Para hacer funcionar las placas Intel Galileo Gen 2 se tuvo que instalar un sistema operativo; Por recomendación del fabricante, se instaló una distribución de debian (Debian 7) la cual proporciona toda una serie de programas y utilidades básicas para poder desempeñar las funcionalidades del proyecto.

En las secciones posteriores se hablará de las características de este sistema operativo y de los paquetes instalados.

3.2.1 Concepto Debian

Debian o Proyecto Debian [5], es una comunidad conformada por desarrolladores y usuarios, que mantienen un sistema operativo GNU. Es un sistema que se encuentra pre compilado, empaquetado y en formato deb para múltiples arquitecturas de computadores con la intención de separar en sus versiones el software libre del no libre. Entre sus trabajadores se tiene la consigna de conseguir que el nombre Debian esté a la altura del nombre Linux.

Su modelo de desarrollo es ajeno a motivos empresariales o comerciales, siendo éste llevado adelante por sus propios usuarios (aunque cuentan con el apoyo de varias empresas). Debian no hace uso comercial de su software y lo pone a disposición de cualquiera en internet (tanto a empresas como particulares) siempre y cuando se respete su licencia original.

Su comunidad de desarrolladores cuenta con la representación de la organización Software in the Public Interest (Software de interés público) la cual es una organización sin ánimo de lucro que brinda apoyo legal para fomentar el software libre. Todo esto surgió tras el fin del patrocinio de la FSF (Free Software Foundation).

3.2.2 Historia de debian

El proyecto Debian fue fundado en el año 1993 por Ian Murdork [5], Después de haber estudiado en la universidad de Purdue. El nombre del proyecto se basa en la combinación de Deborah (del nombre de su entonces novia y posteriormente esposa) y su propio nombre Ian, formando la contracción Debian.

En 1996, Bruce Perens sustituyo a Ian Murdork como el líder del proyecto por sugerencia del desarrollador Ean Schuessler, Bruce Perens además dirigió el proceso de actualización del contrato social de debían y de las pautas del software libre de debían, definiendo los puntos fundamentales para el desarrollo y subdesarrollo de la distribución.

Bruce Perens se retiró en 1998 antes del lanzamiento de la primera versión de debían basada en glibc (versión 2.0) donde el proyecto procedió a buscar nuevos líderes para posteriores revisiones de la versión. Convenientemente fue lanzada durante este periodo con un núcleo no basado en el núcleo de Linux, naciendo así Debian GNU/Hurd.

A finales del 2000, el proyecto realizó el mayor cambio de la estructura de archivos y organización de versiones, se creó un nuevo sistema para la liberación de paquetes software denominado package pools, creando a su vez una rama de prueba estable para futuros lanzamientos.

3.3 Características de Red

Para el proyecto se utilizó una red de clase C la cual proporciona un direccionamiento suficientemente amplio para poder comunicar las placas Intel Galileo entre ellas.

Como se puede apreciar en la figura 27, tenemos conectadas 4 placas Intel Galileo Gen 2 a un router cisco 1921 para poder controlar el tráfico de red que fluye entre ellas. Cabe destacar que el único experimento donde se llegaron a utilizar las 4 placas fue en el algoritmo de multiplicación de matrices ya que se deseaba comparar los resultados obtenidos con los del proyecto de investigación realizado por los ingenieros ***Sherihan Abu ElEnin y Mohamed Abu ElSoud*** en su proyecto de investigación ***Evaluation of Matrix Multiplication on an MPI Cluster*** donde llegaron a utilizar hasta 12 núcleos en un ordenador IBM con procesador Intel Pentium IV con 2.4 GHz y 256 MB de memoria SDRAM. En el resto de experimentos solo se llegó a utilizar 3 placas ya que nos parecieron suficientes los resultados obtenidos donde demostramos que trabajar con programación paralela es más eficiente que trabajar de forma iterativa.

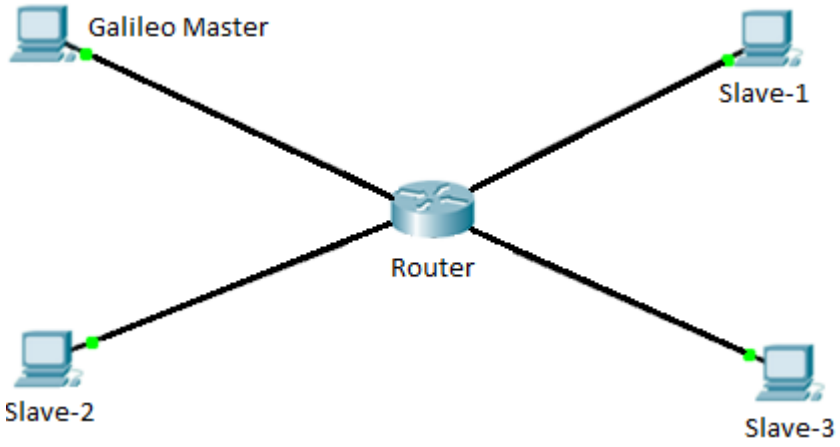


Ilustración 27. Montaje experimental

El direccionamiento seleccionado para las placas fue el siguiente:

Hostname	Dirección IP	Máscara de red	Gateway
Galileo Master	10.10.10.240	255.255.255.0	10.10.10.1
Galileo slave-1	10.10.10.241	255.255.255.0	10.10.10.1
Galileo slave-2	10.10.10.242	255.255.255.0	10.10.10.1
Galileo slave-3	10.10.10.243	255.255.255.0	10.10.10.1
Router	10.10.10.1	255.255.255.0	N/A

Tabla 3. Direccionamiento de red.

3.3.1 Network Address Translation (NAT)

Para poder instalar todo el software necesario y poder trabajar desde internet (como toda placa diseñada para el IoT), se implantó un NAT que permitió hacer traducciones de IP pública a IP privada. Gracias a esta traducción y únicamente conociendo la IP pública del proyecto (o su resolución de nombre), se puede asignar trabajo a las placas y obtener un resultado sin la necesidad de conocer cómo está el Grid configurado internamente.

En la siguiente imagen mostramos cómo funciona el NAT del proyecto.

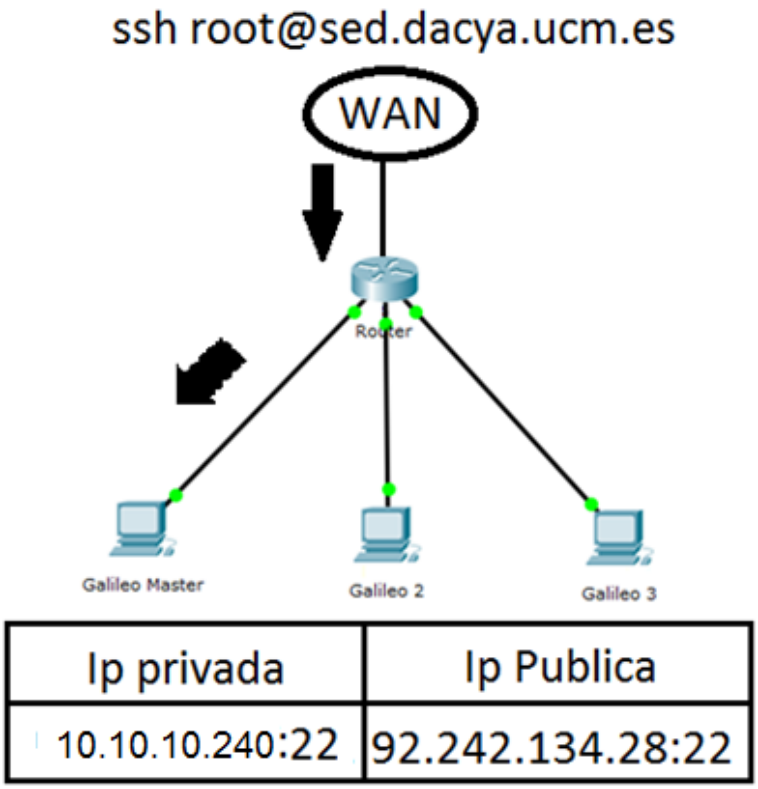


Ilustración 28. Ejemplo conexión desde internet

Este ejemplo muestra una conexión ssh entre un equipo desde internet hacia nuestra ip pública. El router realiza una traducción estática (realiza un volcado puerto a puerto, desde puerto 1 hasta el 65535) a la Galileo master. De esta manera trabajamos como cualquier componente de IoT, tenemos un elemento central donde obtenemos las tareas a realizar e internamente se realizan estos trabajos; ya finalizadas las tareas devolvemos el resultado y el grid queda a la espera de nuevas tareas.

3.3.2 Ataques de fuerza bruta

En criptografía, se le denomina ataque de fuerza bruta a la forma de conseguir una clave probando todas las combinaciones posibles hasta dar con ella y permitir el acceso.

A lo largo de la vida del proyecto, se sufrió una gran cantidad de ataques de esta naturaleza para poder tener acceso a las placas Intel Galileo Gen 2 (en una ocasión consiguieron acceder a ellas). El origen de estas IPs se pueden ver en el fichero Linux /var/log/secure. Realizando un rastreo mediante la url de la ripe.net [23] se determinó que un gran número de ataques recibidos provenían de China, para poder solventar este problema se formatearon las placas y se procedió a cambiar las contraseñas. Escogimos poner contraseñas más fuertes (alfanuméricas de 32 elementos y una combinación entre letras mayúsculas, minúsculas y números) y tener una red más privada, utilizamos PAM de Linux (autenticación más fuerte) y utilizamos el firewall de Linux (iptables) para que sólo admitiera las IPs públicas que el proyecto requería.

3.4 Instalación y ejecución de programas en MPI

Una vez escogido los algoritmos, era necesario saber cómo funciona MPI. Como ya se explicó en capítulos anteriores, MPI es un estándar [8] que facilita la realización de paso de mensajes entre aplicaciones paralelas basadas en paso de mensajes. Esto es una gran ventaja, ya que no es necesario tener un conocimiento exacto del hardware a utilizar. Esta idea se puede entender mejor en la ilustración 29.

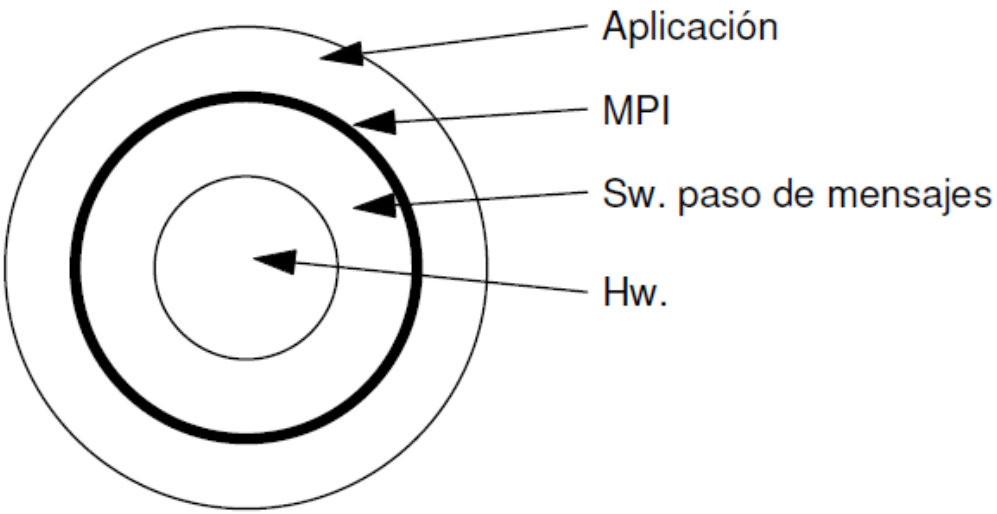


Ilustración 29. Como funciona MPI [8]

Como se puede observar en la ilustración 29, cuando se desarrolla una aplicación en MPI, no es necesario conocer con exactitud el hardware del sistema, ya que MPI realiza muchas de estas tareas necesarias para la correcta ejecución independiente del hardware donde es ejecutado.

Una vez comprendido esto, es necesario entender cómo MPI realiza las comunicaciones entre diferentes procesos. En la siguiente ilustración damos una pequeña explicación de esto.

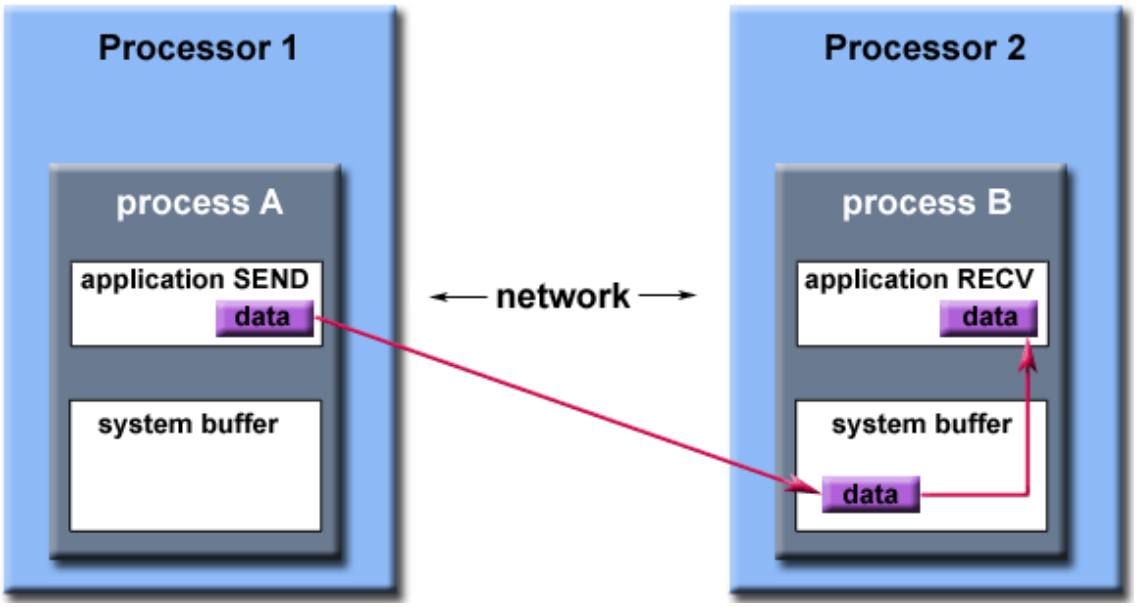


Ilustración 30. Intercambio de datos en MPI [8]

MPI observa qué variables se deben transferir a otros procesos, tal como se observa en la ilustración 30. Diferentes procesos, en diferentes placas, tienen que realizar tareas diferentes, pero si el proceso A no puede culminar su tarea hasta que el proceso B termine, una vez el proceso B termine su tarea, éste debe regresar los datos procesados al proceso A para que éste pueda terminar con su trabajo.

Un ejemplo de como el nodo principal recupera los datos está en la ilustración número 31. Una vez el nodo principal termina de recolectar los datos, puede tratar la solución completa y terminar la ejecución del problema.

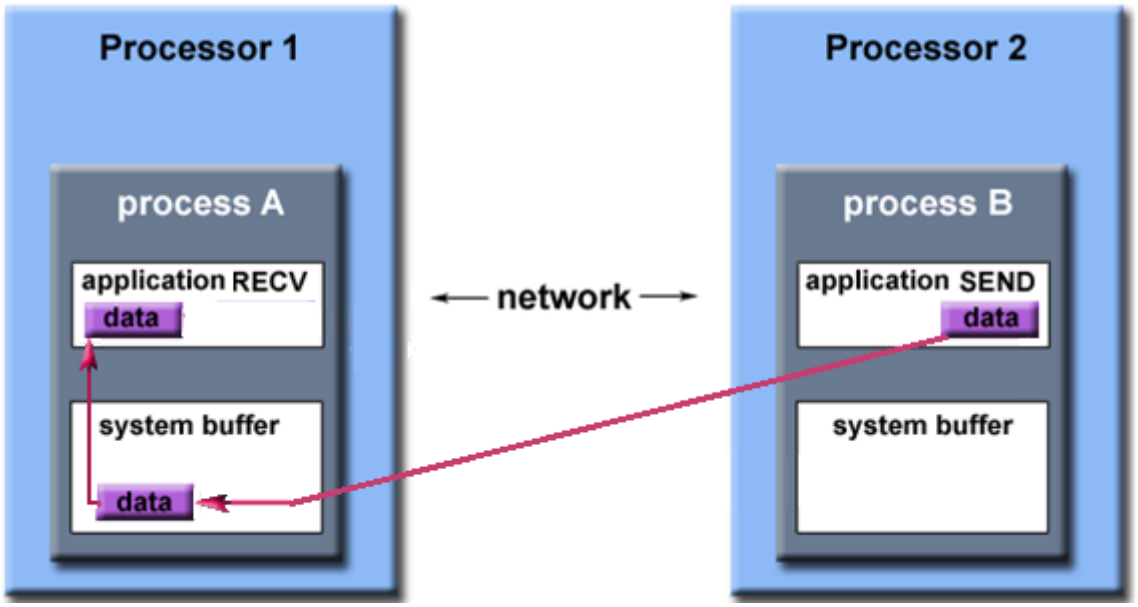


Ilustración 31. Recuperación de datos [8]

Una vez entendido el funcionamiento de MPI, es necesario instalar una serie de utilidades en el sistema. A continuación se dan los pasos básicos para poder instalar MPI en la plataforma Debian 7.

Se deben de instalar los siguientes paquetes para el correcto funcionamiento de MPI:

- **openmpi-bin:** instala librerías y comandos que permiten la ejecución de códigos en paralelos (mpirun).
- **openssh-client, openssh-server:** Programas de comunicación (rutinas de control y presentación) entre procesos.
- **libopenmpi-dbg:** Generador de información de depuración para MPI.
- **libopenmpi-dev:** Necesario para el desarrollo de programas basados en MPI (mpicc, mpicxx, etc).

Estos tienen que ser instalados de la forma que se puede observar en la ilustración 32.

```
sudo apt-get install openmpi-bin openssh-client openssh-server libopenmpi-
dbg libopenmpi-dev
```

Ilustración 32. Instalación de MPI

Aunque no se explicó anteriormente, las comunicaciones entre diferentes procesos en diferentes placas, se realizan mediante una relación de confianza en SSH (Secure Shell).

Una vez realizadas estas instalaciones, es necesario incluir las variables de entorno necesarias en el PATH del sistema para que MPI funcione correctamente. Para sistemas que usen bash, se tendrán que utilizar el comando export.

A continuación, damos un ejemplo de configuración de las variables de entorno.

```
export PATH="$PATH:/home/$USER/.openmpi/bin";
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/home/$USER/.openmpi/lib/"
```

Ilustración 33. Configuración de variables de entorno

Sin estas configuraciones, no se podrá trabajar con la herramienta MPI.

Otro aspecto importante para que MPI pueda trabajar de forma correcta, es que es necesario establecer una relación de confianza entre las diferentes placas mediante SSH.

Para esto se necesita configurar la clave pública (RSA, Rivest, Shamir y Adleman), y luego compartirla con las diferentes placas.

El proceso es sencillo y se describe a continuación; para generar una clave pública, se tiene que proceder con el siguiente comando:

- **ssh-keygen –b 1024 –t rsa**

De esta manera, se crea una clave pública de tamaño 1024 y de tipo RSA. Este tipo de claves, se guarda en el directorio HOME del usuario root, en la carpeta .ssh y con nombre del fichero id_rsa.pub.

Una vez realizada esta tarea, es necesario compartir esta clave con las diferentes placas en el grid, un ejemplo de copiado podría ser el siguiente

- **ssh-copy-id root@<ip placa destino>**

Cuando se termine el proceso de compartir las claves publicas entre las diferentes placas, ya se puede compilar y ejecutar diferentes programas en el grid.

Cabe destacar que es necesario, una vez programado el algoritmo en MPI, copiar el ejecutable entre las diferentes placas, ya que si no, éste no podría trabajar de la forma esperada.

3.4.1 Compilar programas en MPI

Para poder compilar los programas en MPI [8] es necesario un compilador especial que entienda la sintaxis de este estándar. Pero a pesar de ser especial, se fundamenta mucho en la forma en que C compila sus programas. A continuación damos un ejemplo de cómo se tendría que compilar un programa en MPI:

- **mpicc** < Nombre del programa en MPI > **-o** < Nombre del ejecutable>

3.4.2 Ejecutar programas en MPI

Para ejecutar los programas en MPI [8], es necesario comprender sus funcionalidades. A continuación mostramos un listado de las diferentes configuraciones para poder sacar el mejor provecho.

- **mpirun** [options] **<program>** [<args>]

La cantidad de argumentos es demasiado grande como para explicar todas en el presente documento, en lugar de eso explicaremos las utilizadas para poder realizar el presente proyecto de investigación [8].

- **-H**, aquí podemos definir los host que van a ejecutar el programa MPI.
- Es posible especificar un fichero de host para resumir el comando en el caso de que sean muchas placas, con la opción -hostfile
- **-np**, especificamos la cantidad de procesos que se van a crear para el programa a ejecutar.
- **-bynode**, con esta opción se le especifica la cantidad de procesos que van a correr por cada placa. En nuestro caso sería igual a 1.

- **-out-put-filename**, escribimos el resultado del comando en el fichero especificado.
- **--allow-run-as-root**, Es aconsejable utilizarlo cuando se pretende lanzar ejecuciones de MPI desde un script, de esta manera el sistema cree que el usuario root ha ejecutado el script.

Para entender mejor estas opciones, en la siguiente ilustración mostramos un ejemplo (script usado para ejecutar el quicksort) de cómo se ejecutan los programas MPI.

Para ejecutarlos lo hacemos con un script llamado compilar que tiene la siguiente estructura.

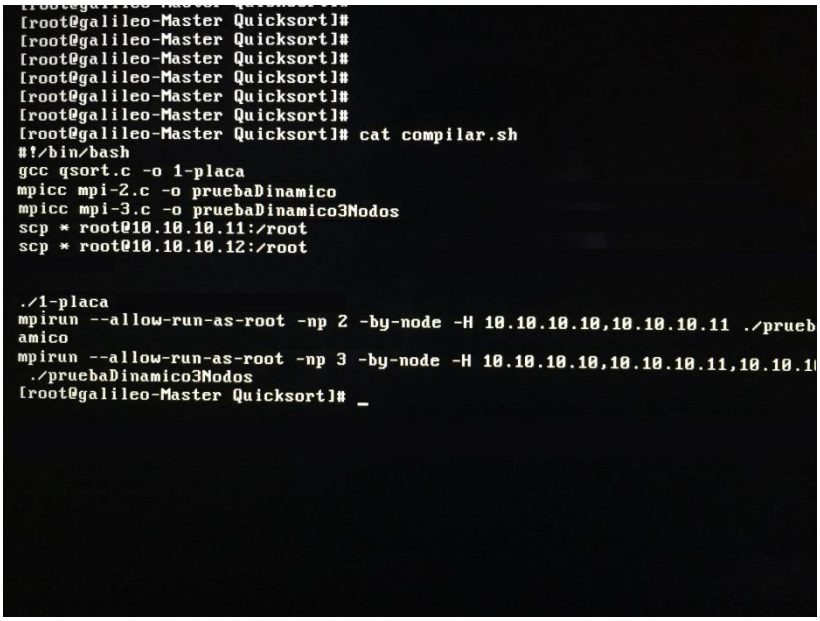


Ilustración 34. Script de ejecución.

De esta manera se hace más cómoda la ejecución de los programa ya que éste se encarga de transferir los ejecutables a las diferentes placas y ejecutarlos. Una vez programados los algoritmos y comprobado su correcto funcionamiento, era necesario programar 2 scripts que nos permitiera la ejecución de los algoritmos una cantidad elevada de ejecuciones.

El primer script se le denomino repeticiones.sh, el cual permite la ejecución de los algoritmos unas 1000 veces (se consideró que para explorar y verificar la exactitud de los resultados obtenidos era necesaria de una cantidad alta de ejecuciones),

adicionalmente se creó un fichero denominado i.log (donde el elemento i representa la cantidad de placas utilizadas) en el que se resguardaban los resultados obtenidos por los programas ejecutados en el script.

El segundo script se le denomino media.sh en el que se calcula la media aritmética de los valores obtenidos en el fichero i.log.

Para finalizar se puede observar el código del script *repeticiones.sh* en la siguiente ilustración.

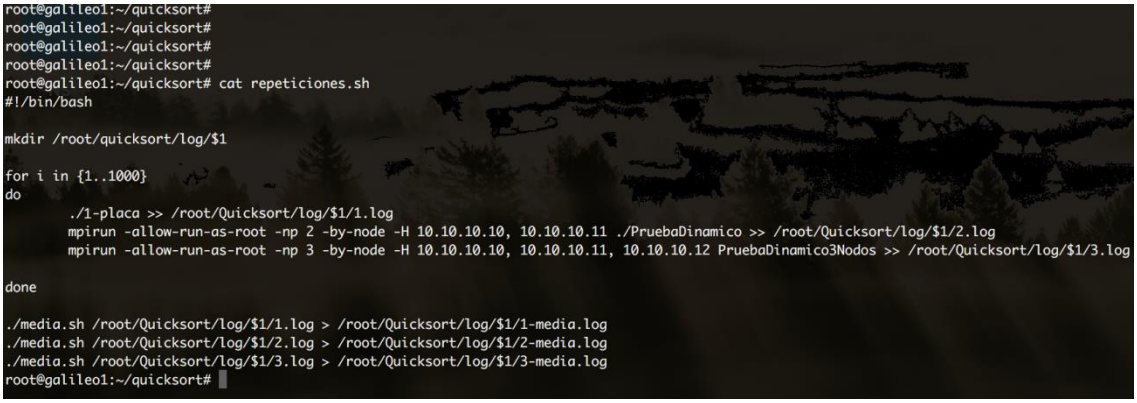


Ilustración 35.Script de ejecución y media.

Adicionalmente, es necesario configurar en el sistema la hora de comienzo de ejecución del script ejecuciones.sh, debido a que son tareas que tardan una cantidad de tiempo considerable, se decidió utilizar el servicio crontab de debían para programar la ejecución del script a una determinada hora. Se optó por ejecutar las tareas a las 00:00 y recibir los resultados de dicha ejecución por correo electrónico (la explicación de cómo enviar resultados mediante correo electrónico serán abordados en el sub-capítulo 3.4.5).

3.4.3 Calcular tiempos de ejecución

Para el cálculo de los tiempos de ejecución, se optó por utilizar unas bibliotecas estándares de C denominadas **time.h** y **sys/time.h**. Este estándar contiene funciones para manipular y formatear la fecha y hora del sistema.

- Su funcionamiento es bastante sencillo. Para comenzar, se tiene que declarar la variable (variables de tipo timeval) para conocer el instante en que se quiere comenzar a hacer las mediciones.
- Posteriormente, se debe proceder a ejecutar todo el algoritmo.
- Para finalizar, se vuelve a realizar una medición en el tiempo para saber el tiempo exacto en el que termina la ejecución del algoritmo. Teniendo los datos de comienzo y fin, se realiza la diferencia entre ellos para conocer el tiempo real.

A continuación se muestra un ejemplo de cómo calcular los tiempos de ejecución de un problema.

1. Lo primero es declarar las variables de tipo timeval, estas variables nos permiten conocer el tiempo exacto en el que un programa comienza y finaliza su ejecución, posteriormente se debe delimitar el código del programa del que se desea conocer su tiempo de ejecución. Un ejemplo de cómo se realizaría dicha acción seria el siguiente:

```
Struct timeval t_ini, t_fin;
double secs;

Gettimeofday (&t_ini, NULL);
/* ...Programa a calcular... */
Gettimeofday (&t_fin, NULL);
```

2. Lo segundo es declarar una función auxiliar que nos permita realizar la diferencia entras las variables comienzo y fin.

```
double timeval_diff (struct timeval *a, struct timeval *b)
{
  return
    (double) (a -> tv_sec + (double) a -> tv_usec/1000000) -
    (double) (b -> tv_sec + (double) b -> tv_usec/1000000);
}
```

Gracias a esta función se puede determinar el tiempo en segundos que ha tardado un programa.

En los siguientes sub-capítulos se expondrán unas series de herramientas utilizadas que ayudaron a facilitar el trabajo. Estas en un principio no son necesarias para el desarrollo del proyecto, pero sí generan un plus añadido como poder enviar resultados obtenidos a través de correos electrónicos.

3.4.4 Configuración de resolución de nombres (DNS)

Un sistema de nombres de dominio (por sus siglas en inglés, *Domain Name System*) es un servicio de nomenclatura jerárquico que permite asociar nombres de dominios a una determinada IP, su función más importante es "traducir" nombres para determinados equipos interconectados en una red. En la siguiente ilustración se puede apreciar mejor cómo funciona dicho servicio.

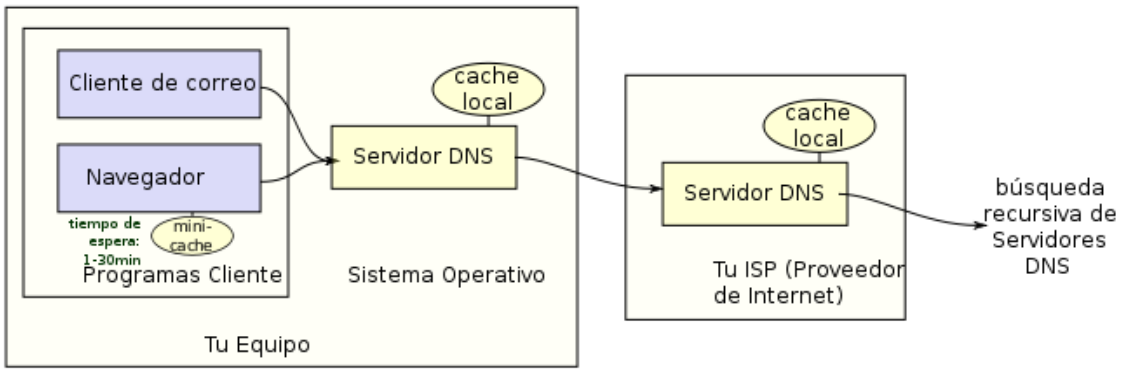


Ilustración 36.Funcionalidad de servicio DNS.

En nuestra propuesta fue incluido ya que es más fácil comunicar los diferentes procesos entre las placas por sus nombres que por sus respectivas direcciones IP.

Un ejemplo de cómo funciona puede ser el presentado en la ilustración 37 donde se interesa conocer la ip asociada a la placa slave-1.

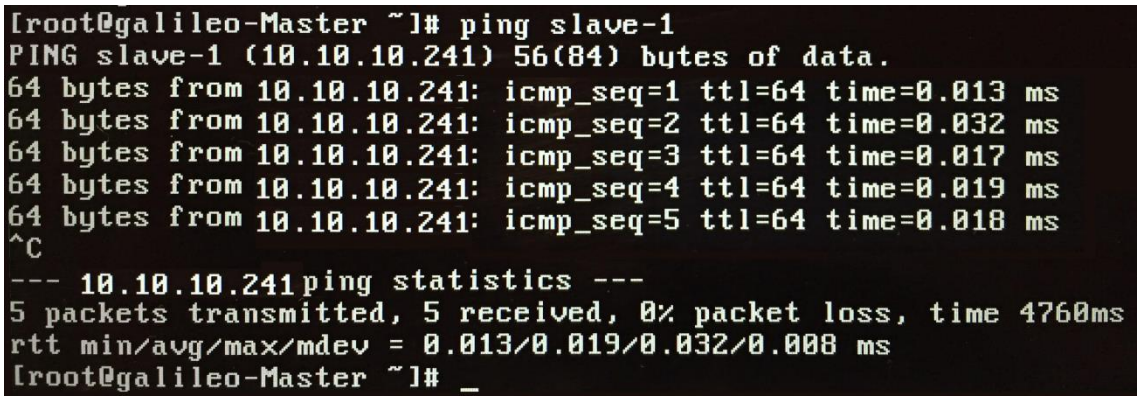
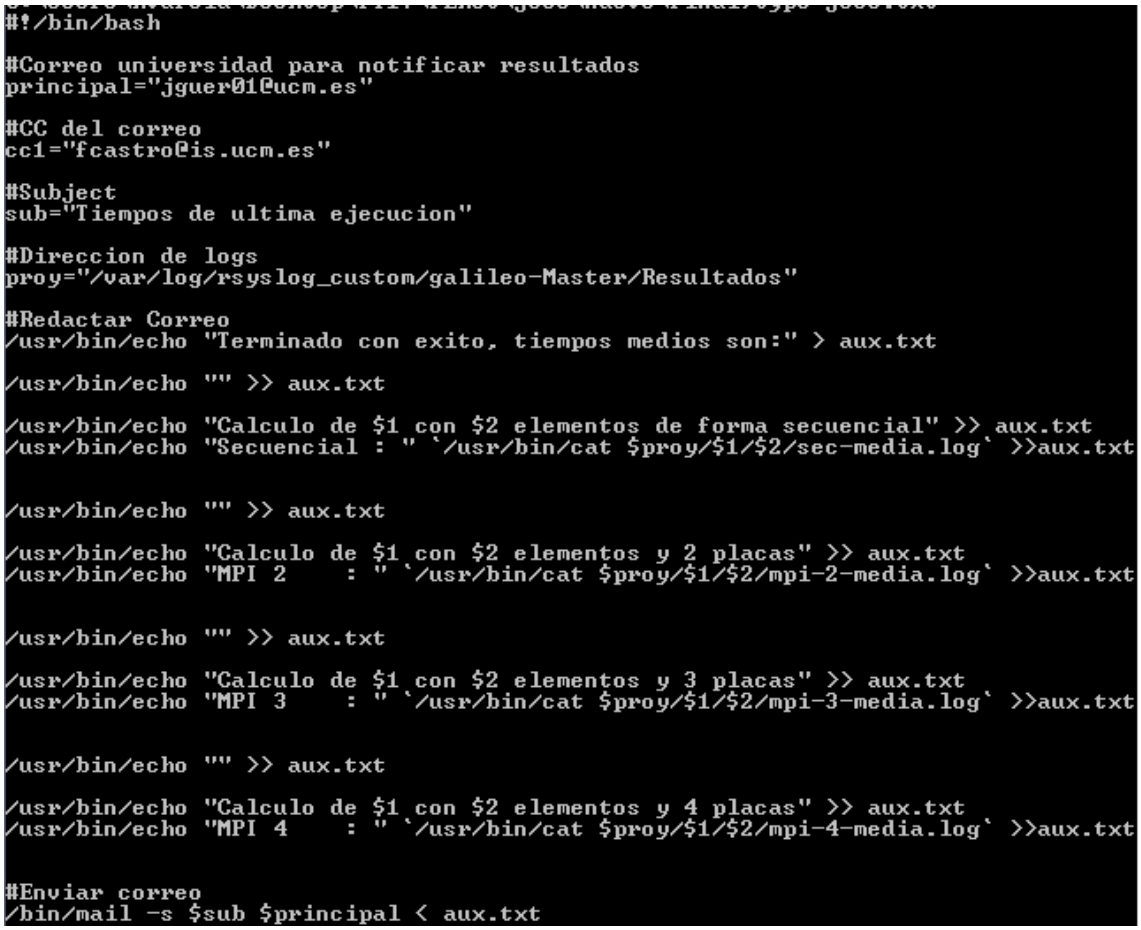


Ilustración 37. Resolución de nombre a dirección IP.

De esta forma es más fácil generar scripts ya que nos permite comunicarnos con las diferentes placas sólo sabiendo su nombre de dominio en lugar de la IP asociada a ésta. Para finalizar con el capítulo 3, se expondrá la última utilidad desarrollada que nos permite notificar los resultados de forma más sencilla mediante correo electrónico.

3.4.5 Envió de resultados por correo

Para obtener los resultados de forma más sencilla y sin la necesidad de tener que acceder a las placas, se consideró oportuno implementar un script que fuera capaz de enviar los resultados de las diferentes ejecuciones por correó. A este script lo denominamos notificar.sh y se puede observar su estructura en la ilustración 38.



```
#!/bin/bash

#Correo universidad para notificar resultados
principal="jguer01@ucm.es"

#CC del correo
cc1="fcastro@is.ucm.es"

#Subject
sub="Tiempos de ultima ejecucion"

#Direccion de logs
proy="/var/log/rsyslog_custom/galileo-Master/Resultados"

#Redactar Correo
/usr/bin/echo "Terminado con exito, tiempos medios son:" > aux.txt

/usr/bin/echo "" >> aux.txt

/usr/bin/echo "Calculo de $1 con $2 elementos de forma secuencial" >> aux.txt
/usr/bin/echo "Secuencial : " `/usr/bin/cat $proy/$1/$2/sec-media.log` >>aux.txt


/usr/bin/echo "" >> aux.txt

/usr/bin/echo "Calculo de $1 con $2 elementos y 2 placas" >> aux.txt
/usr/bin/echo "MPI 2    : " `/usr/bin/cat $proy/$1/$2/mpi-2-media.log` >>aux.txt


/usr/bin/echo "" >> aux.txt

/usr/bin/echo "Calculo de $1 con $2 elementos y 3 placas" >> aux.txt
/usr/bin/echo "MPI 3    : " `/usr/bin/cat $proy/$1/$2/mpi-3-media.log` >>aux.txt


/usr/bin/echo "" >> aux.txt

/usr/bin/echo "Calculo de $1 con $2 elementos y 4 placas" >> aux.txt
/usr/bin/echo "MPI 4    : " `/usr/bin/cat $proy/$1/$2/mpi-4-media.log` >>aux.txt


#Enviar correo
/bin/mail -s $sub $principal < aux.txt
```

Ilustración 38. Notificar resultados por correo.

Su ejecución es muy sencilla y sigue los siguientes pasos:

- Primero guardamos en variables el destinatario del correo y los posibles CC a los que se les desea enviar el correo. A estas variables las denominamos principal y cc1.
- Se genera una variable denominada sub donde se guarda el asunto del correo.
- Una última variable denominada proy donde guarda el directorio donde están alojados los logs (el argumento $1 representa el tipo de problema y el argumento $2 la cantidad de elementos que se utilizaron para el cálculo)
- Adicionalmente, se utilizó un archivo temporal denominado aux.txt donde se genera el cuerpo del correo.

- Para finalizar se ejecuta el comando mail para enviar el correo.

Capítulo 4. Resultados

En el presente capítulo se analizarán los resultados obtenidos por los diferentes algoritmos, los cuales vendrán expuestos mediante gráficas. Cabe destacar que los tiempos expuestos en el presente capitulo han sido calculados por la siguiente formula:

$$\boldsymbol{Tiempo_{Ejecución} = Tiempo_{Comunicación} + Tiempo_{Computo}}$$

A continuación se dará una breve explicación:

- $\boldsymbol{Tiempo_{Comunicación}}$: equivale al tiempo que tardan las placas en comunicar los datos, este tiempo se podría representar con la siguiente formula:

  $$\boldsymbol{Tiempo_{Comunicación} = Tiempo_{Enviar\ datos} + Tiempo_{Recibir\ resultados}}$$

  Como se puede observar, se tienen en cuenta ambos tiempos debido a que la placa principal envía los datos iniciales a las placas esclavas y una vez está resuelto su parte del problema éstas regresan su resultado a la placa principal.

- $\boldsymbol{Tiempo_{Computo}}$: es el tiempo en que tardan en ejecutar el algoritmo.

El tiempo de cómputo suele ser estable debido a que son algoritmos bastante deterministas, por lo que la variación de un resultado a otro varía casi en su totalidad por el tiempo de las comunicaciones. Al ser placas con un único procesador y thread, estas tienen que detener procesos para analizar las tramas Ethernet que les llegan por la tarjeta de red por lo que se puede llegar a generar retardos no deseados.

También se debe destacar que el speedup o beneficio (en tiempo de ejecución) de los resultados obtenido es calculado mediante la siguiente formula:

$$\boldsymbol{Speedup} = \frac{\boldsymbol{Tiempo_{Ejecución}\ \ Secuencial}}{\boldsymbol{Tiempo_{Ejecución}\ \ Paralelo}}$$

Los tiempos pueden ser explicados de la siguiente forma:

- $\boldsymbol{Tiempo_{Ejecución}\ \ Secuencial}$: Es el tiempo en el que tarda 1 placa en resolver un algoritmo.
- $\boldsymbol{Tiempo_{Ejecución}\ \ Paralelo}$: Es el tiempo en el que tardan N placas en resolver un algoritmo.

En los siguientes sub-capítulos se analizaran los resultados obtenidos por los diferentes algoritmos con la finalidad de estudiar el beneficio total al trabajar con MPI. Adicionalmente, en el sub-capítulo 4.5 se expondrán los recursos invertidos (CPU y memoria RAM) por cada algoritmo en la placa Intel Galileo Gen 2.

4.1 Resultados algoritmo Serie x^2

En la presente sub-sección se analizaran los resultados obtenidos por el primer algoritmo. Como se expresó en capítulos anteriores fue el algoritmo más sencillo de programar y sus resultados fueron los más rápidos de obtener. En la siguiente ilustración se puede apreciar de forma muy resumida el comportamiento de los tiempos de ejecución del algoritmo en las placas Intel Galileo Gen 2.

Ilustración 39. Tiempos de ejecución del algoritmo Serie x^2

Para entender mejor los resultados expuestos, se expondrán una serie de graficas normalizadas donde se explica mejor el comportamiento del algoritmo.

1. Para comenzar, se normalizarán los resultados obtenidos (al tiempo de ejecución resultante de 1 placa) en la adición de los 1000 primeros números naturales.

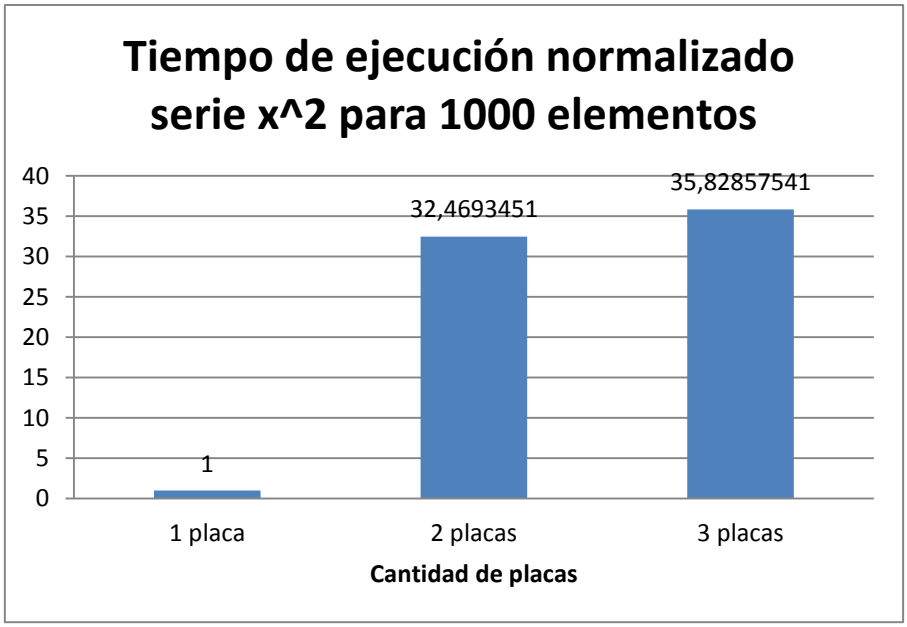


Ilustración 40 . Normalización de tiempos de ejecución del algoritmo Serie x^2 para 1000 elementos naturales.

Como se pudo observar en la gráfica normalizada, trabajar con MPI para tamaños de problemas muy pequeños se genera un retardo demasiado grande (con 2 placas de 32 veces peor y con 3 placas de casi 36) en comparación al trabajar con una única placa, esto se debe a la forma en que trabaja MPI, como se explicó anteriormente, MPI es una interfaz con métodos propios los cuales utilizan las librerías de C para poder realizar sus operaciones, por lo que se debe sumar este retardo, además se debe de tener en cuenta que MPI debe comunicar datos entre las diferentes placas siendo este tiempo de transmisión muy superior si lo comparamos con el tiempo en que tarda la CPU al buscar un dato en la memoria cache.

Para problemas con cálculos tan pequeños MPI no es una buena opción ya que suele tardar más que si se trabaja de forma secuencial (algunas de sus limitaciones ya fueron mencionadas en capítulos anteriores) por lo que se debe utilizar cuando el tamaño del problema es considerablemente grande. Esto se debe a que se ve muy limitado por la velocidad de transferencia de los datos por lo que para realizar pequeños cálculos no compensa dividir el problema en unidades más pequeñas.

2. El siguiente punto escogido fue un tamaño de 10 Millones de elementos, a pesar de que el tamaño del problema se podría considerar grande, aun el retardo de las comunicaciones es mayor al tiempo que tarda en calcular la placa 1 de forma secuencial por lo que se debe aumentar aún más el tamaño del problema.

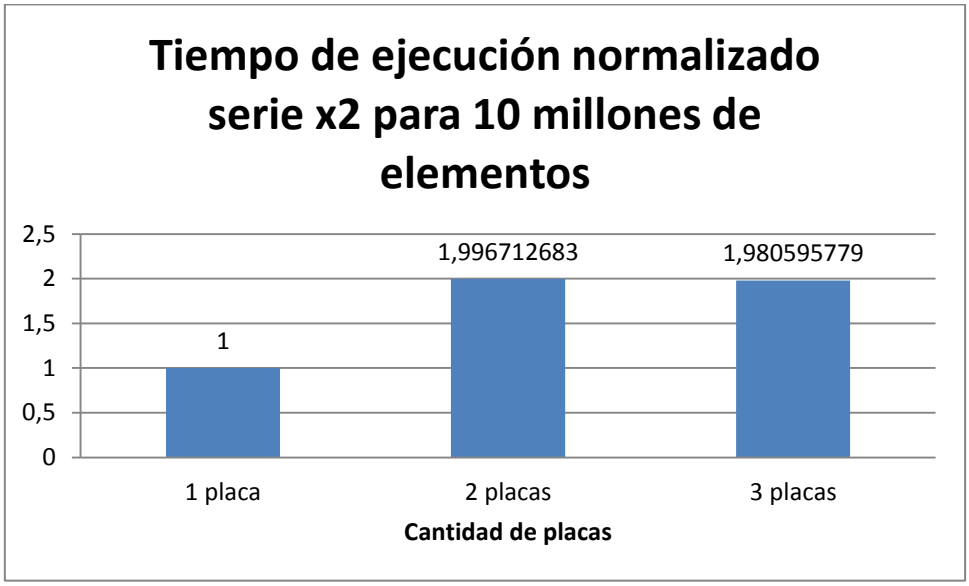


Ilustración 41 Normalización de resultados del algoritmo Serie x^2 hasta 10 millones de elementos.

3. El siguiente punto escogido fue el de 24 millones de elementos, como se puede ver en la siguiente ilustración el trabajar con 3 placas ya se genera un beneficio del casi 8 % mientras que si lo comparamos con el tiempo de 2 placas éste sigue siendo superior a trabajar con una única placa.

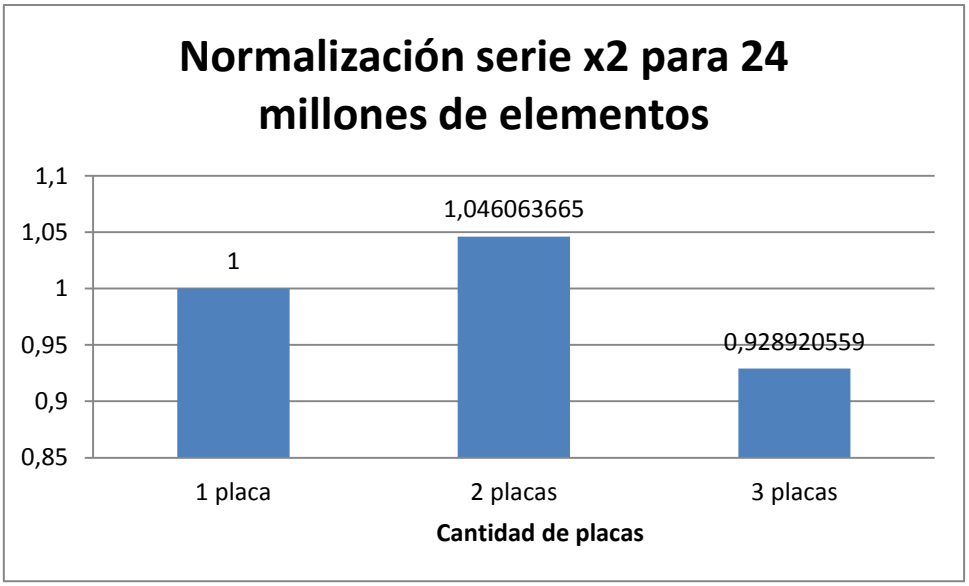


Ilustración 42 Normalización de resultados del algoritmo Serie x^2 hasta 24 millones de elementos.

4. El siguiente punto escogido fue el de 29 Millones de elementos donde se puede observar que el tiempo de ejecución con 2 placas ya es inferior a trabajar con una única placa generando un ahorro del casi 6 %. Además de esto se puede observar que trabajar con 3 placas ya genera un beneficio de casi el 18 % para el mismo tamaño de problema.

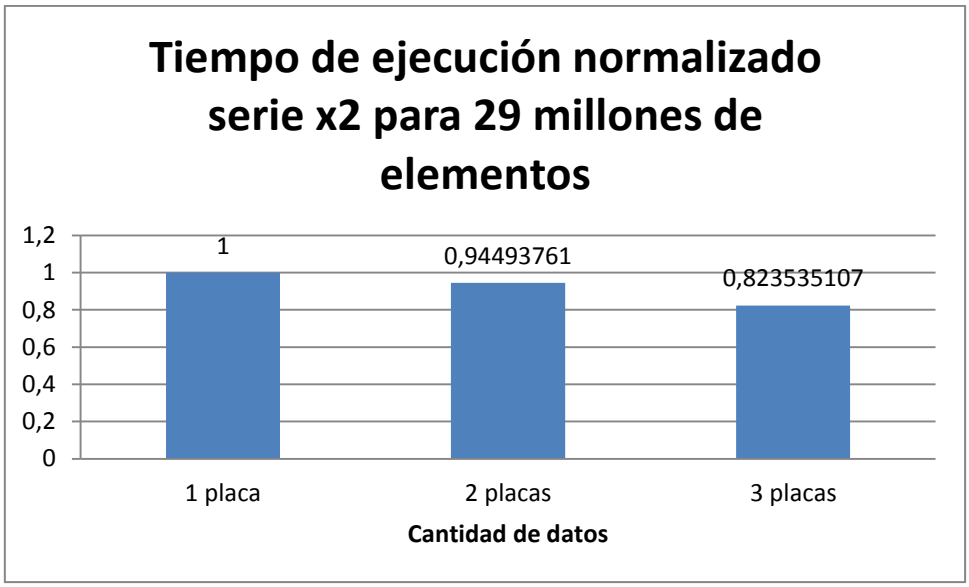


Ilustración 43 Normalización de resultados del algoritmo Serie x^2 hasta 29 millones de elementos naturales.

5. Para finalizar, se ilustra en la siguiente gráfica normalizada la última muestra calculada, la adición de los cuadrados para 99 millones de elementos naturales. Extendiendo el tamaño del problema a un número tan grande se puede apreciar un beneficio enorme si lo comparamos con el tiempo invertido por una única placa.

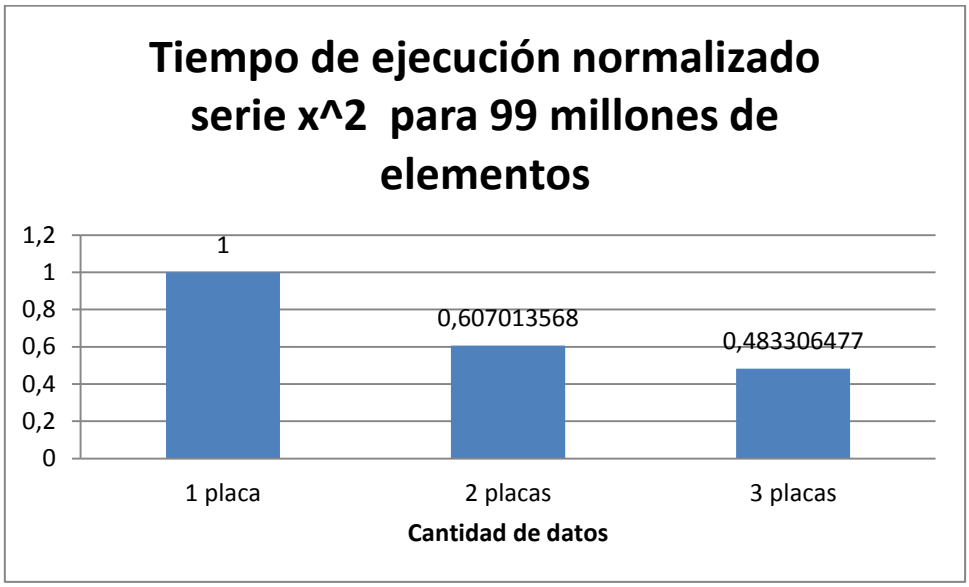


Ilustración 44 Normalización de resultados del algoritmo Serie x^2 hasta 99 millones de elemetos naturales.

Al trabajar con 2 placas se consiguió un beneficio del 39,29 % mientras que con 3 placas fue del 51,66 %. De esta manera ya sabiendo los puntos de corte de cada grafica (aproximadamente en 22 millones de elementos a sumar) se podría generar un tercer programa donde dependiendo del tamaño se utilizara un método u otro para asegurarse que siempre se trabaja de forma óptima.

Con esto se concluyen los resultados para el primer algoritmo, en el siguiente apartado se expondrán los resultados obtenidos por el algoritmo Quicksort.

4.2 Resultados algoritmo Quicksort

En el siguiente sub-apartado se expondrán los resultados obtenidos por el algoritmo de ordenación Quicksort, al igual que el sub-apartado anterior, se comenzará ilustrando los resultados obtenidos de forma genérica para posteriormente ser analizados mediante gráficas normalizadas.

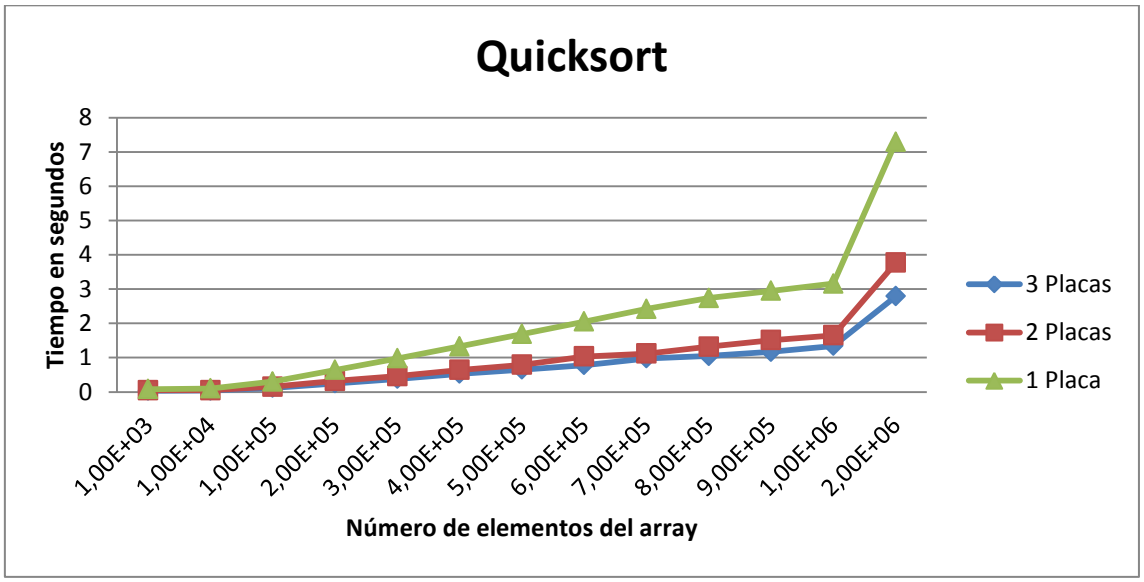


Ilustración 45 Resultados obtenidos por el algoritmo Quicksort.

Como se pudo apreciar en la ilustración 45, se exponen los resultados obtenidos en nuestra propuesta de forma muy simplificada, pero para poder ser analizados más en profundidad se les comparará con otros proyectos de investigación para corroborar la veracidad de dichos resultados, para finalizar se mostrarán también una serie de gráficas normalizadas para explicar mejor el beneficio obtenido mediante MPI.

El primer proyecto que se utilizó como referencia para comprobar la veracidad de los resultados obtenidos fueron los resultados expuestos por la propuesta realizada por ***Kataria P. [9].***

A continuación se expone un pequeño extracto de sus resultados:

Number of processes	Pthread execution time (sec)	MPI execution time (sec)
5	0.23	0.11
7	0.19	0.1
9	0.18	0.08
11	0.16	0.06
13	0.16	0.1
15	0.16	0.15

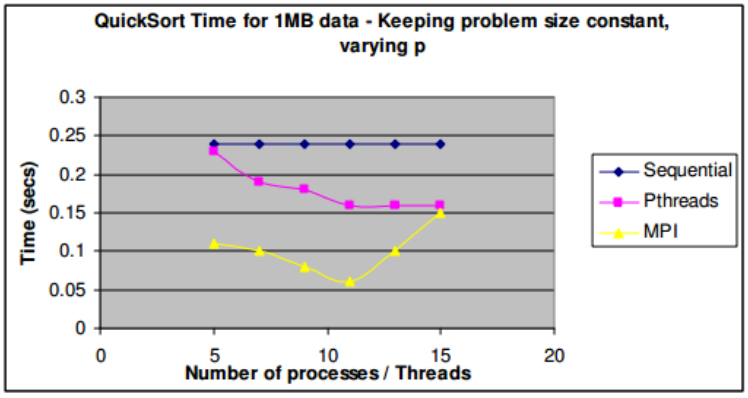


Ilustración 46 Resultados obtenidos por Kataria P [9].

Kataria para ordenar un array de tamaño 1MB (variables de tipo int con un procesador Xeon E5620 de 32 bits, frecuencia de 2.4 GHz) con 5 procesos tardó 0,11 segundos [9], en nuestros experimentos no disponemos de un tamaño exacto de 1MB por lo que se debe de usar como referencia el realizado por el tamaño de 300000 elementos (elementos de tipo int), cada variable int ocupa en memoria lo equivalente a 4 bytes por lo que tener un array de tamaño 300000 sería lo equivalente a tener un tamaño de problema de 1.2 MB (utilizamos la equivalencia de 1 MB igual a 1000 bytes). En nuestro experimento para ordenar dicho array se obtuvieron los siguientes resultados:

	Tamaño del array (300000 elementos ints)
1 placa	0,98 seg
2 placas	0,461 seg
3 placas	0,381 seg

Tabla 4.Resultados obtenidos para ordenar 1.2 MB.

Estos resultados siguen la misma tendencia a los resultados obtenidos por ***Kataria*** donde para un número reducidos de placas (según sus cálculos realizados) los

tiempos de ejecución mejoran hasta llegar a utilizar 11 núcleos, luego de esto se observa una degradación en los resultados debido a que la repartición de trabajo entre los diferentes núcleos es más laboriosa en comparación a la cantidad de trabajo a realizar.

Adicionalmente compararemos nuestros resultados a otro proyecto de investigación, este fue realizado por ***Alaa Ismail El-Nashar [6].*** En su propuesta, ordenaba arrays mediantes diferentes algoritmos de ordenación de los cuales sólo nos centraremos en los resultados obtenidos por el algoritmo quicksort, utilizando MPI en un sistema basado en Windows con 2 cores (Dual-Core con CPU E5500 de 2.8 GHz y 3 GB de memoria RAM) llegó a ordenar un array (datos de tipo int) de $\mathbf{6x10^6}$ de elementos. A continuación se ilustra un pequeño extracto de sus resultados.

Number of cores	Number of processes	Execution time in seconds		
		Bubble sort, 2×10^5 date items	Merge sort, 6×10^6 date items	Quick sort, 6×10^6 Date items
1	1	457.375	5.718	4.437
	2	229.250	5.765	4.500
	4	115.859	5.906	4.562
	8	57.812	6.296	4.859
	16	28.953	6.953	5.421
	32	15.140	8.421	6.890
	64	9.0460	11.687	10.14
2	1	460.796	5.609	4.203
	2	117.546	4.031	3.203
	4	57.937	4.234	3.328
	8	29.109	4.328	3.453
	16	14.646	4.640	3.796
	32	7.625	4.968	4.265
	64	4.687	7.109	6.265

Ilustración 47 Resultados obtenidos por Alaa Ismail El-Nashar [6]

En nuestra propuesta no se consideró oportuno crear más de un proceso por placa como en la propuesta de ***Alaa Ismail El-Nashar*** debido a que la placa Intel Galileo sólo dispone de 1 core y 1 thread, por lo que crear procesos adicionales en nuestra propuesta solo ralentizaría el proceso de ordenación.

Como no podemos comparar nuestros resultados a los obtenidos por ***Alaa Ismail El-Nashar,*** se procederá a utilizar sus resultados como referencia y observar si los resultados obtenidos en nuestra propuesta siguen la misma tendencia.

A continuación se ilustrara una serie de graficas normalizadas para analizar las tendencias de los tiempos de ejecución en las placas Intel Galileo Gen 2 para el algoritmo Quicksort.

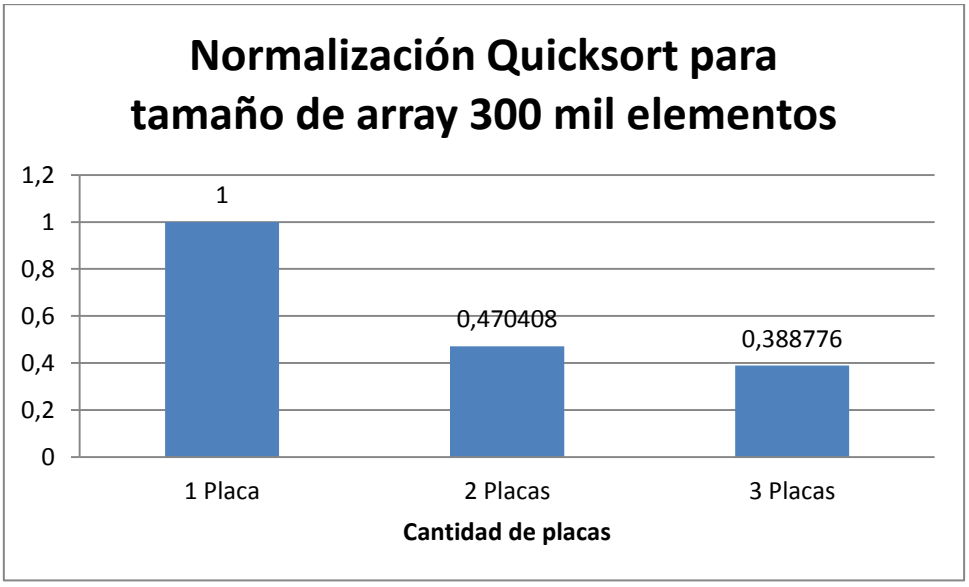


Ilustración 48 Normalización de resultados del algoritmo Quicksort para 300 mil elementos

Como se puede observar en la ilustración 48, para ordenar un array de 300 mil elementos (o lo equivalente a ordenar datos de tamaño 1.2 MB) con 3 placas se obtiene un beneficio del 61,12 %, mientras que se ordena con 2 placas se obtiene del 52,95 %.

Para finalizar, se expondrán los resultados obtenidos para el último experimento realizado, el de un array con tamaño de 2 millones de elementos, lo que representa en memoria unos 8MB.

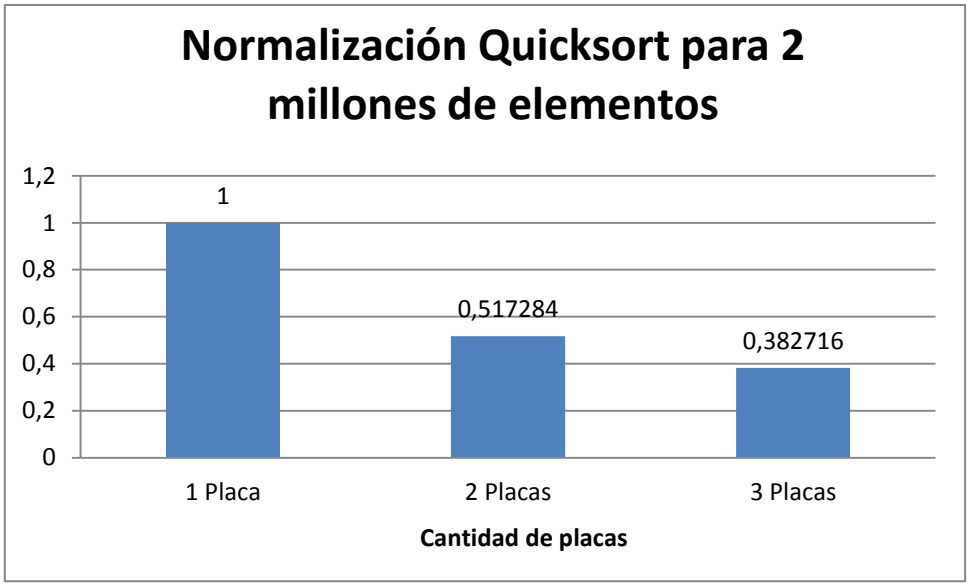


Ilustración 49 Normalización de resultados del algoritmo Quicksort para 2 millones de elementos.

Para este tamaño de problema se obtuvo un beneficio con 3 placas de 61,72 % mientras que para 2 placas uno del 48,27 %, cabe destacar que al no disponer de suficientes placas, no se puede afirmar hasta qué punto es bueno seguir distribuyendo trabajo entre las diferentes placas.

4.3 Resultados algoritmo Multiplicación de matrices

En el presente sub-capítulo se expondrán los resultados obtenidos por el algoritmo de multiplicación de matrices, como en los sub-capítulos anteriores, nuestros resultados serán contrastados a resultados obtenidos por otros proyectos de investigación, en esta ocasión serán comparados por el proyecto realizado por ***Sherihan Abu ElEnin y Mohamed Abu [18].*** En su proyecto utilizaron un equipo Intel Pentium IV con un procesador de 2.4 GHz y 256 MB de memoria SDRAM.

Para comparar los resultados de forma más sencilla, se han replicado las dimensiones de las matrices para poder comparar nuestros resultados a los expuestos por ***Sherihan Abu ElEnin y Mohamed Abu ElSoud***.

4.3.1 Matriz de tamaño 256 x 256

A continuación se expone un extracto de los resultados obtenidos por ***Sherihan Abu ElEnin y Mohamed Abu ElSoud:***

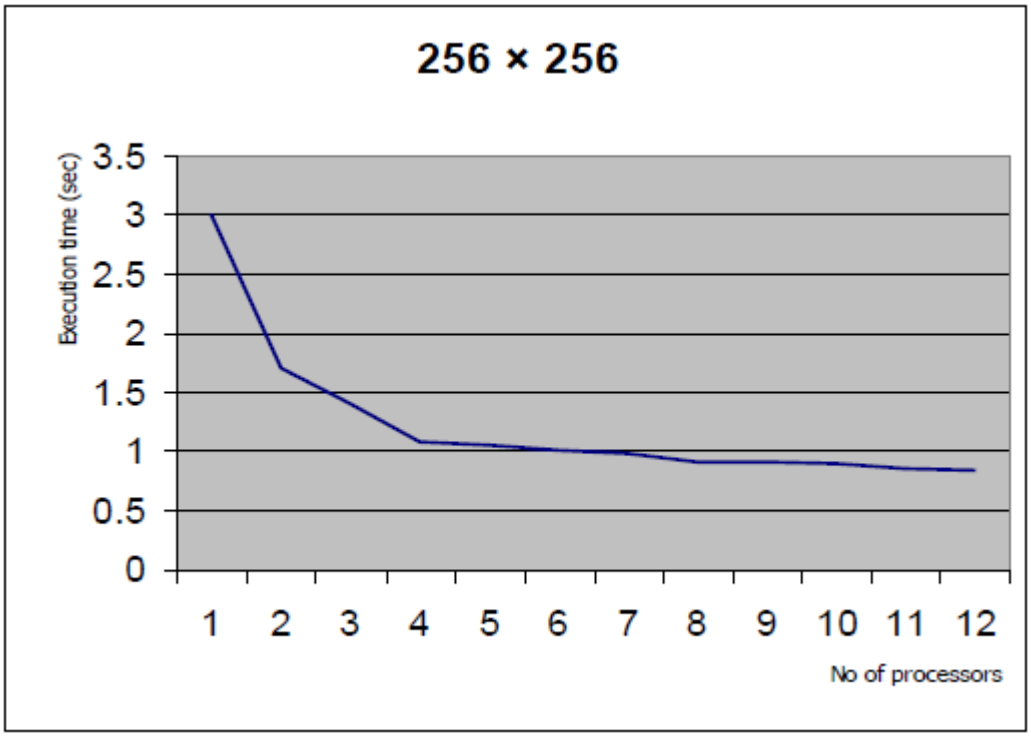


Ilustración 50. Resultados de la multiplicación de matrices 256 x 256 realizados por Sherihan Abu ElEnin y Mohamed Abu ElSoud [18].

Como se puede ver en sus resultados, el tiempo de cálculo se reduce entre más se paralelice el trabajo, pero cabe anotar que el beneficio obtenido va disminuyendo entre más procesos se añadan al programa, por ejemplo, el beneficio de dividir el programa entre 8 procesos en comparación a 12 son casi iguales por lo que la inversión realizada para ese poco tiempo de mejora no compensa al coste en dinero invertido.

A continuación estos resultados serán comparados con los obtenidos en nuestra propuesta:

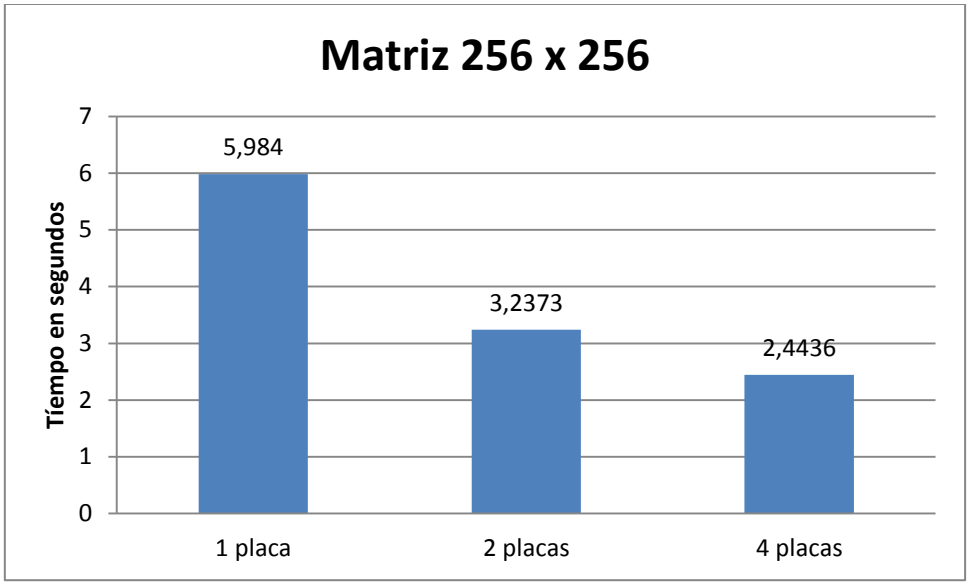


Ilustración 51 Resultados de la multiplicación de matrices 256 x 256.

Como se pudo observar en la ilustración anterior, la tendencia es decreciente según se aumenta el número de procesadores (en nuestro caso placas). Para cuantificar el beneficio obtenido en nuestros resultados se expondrá a continuación una gráfica normalizada para esta matriz.

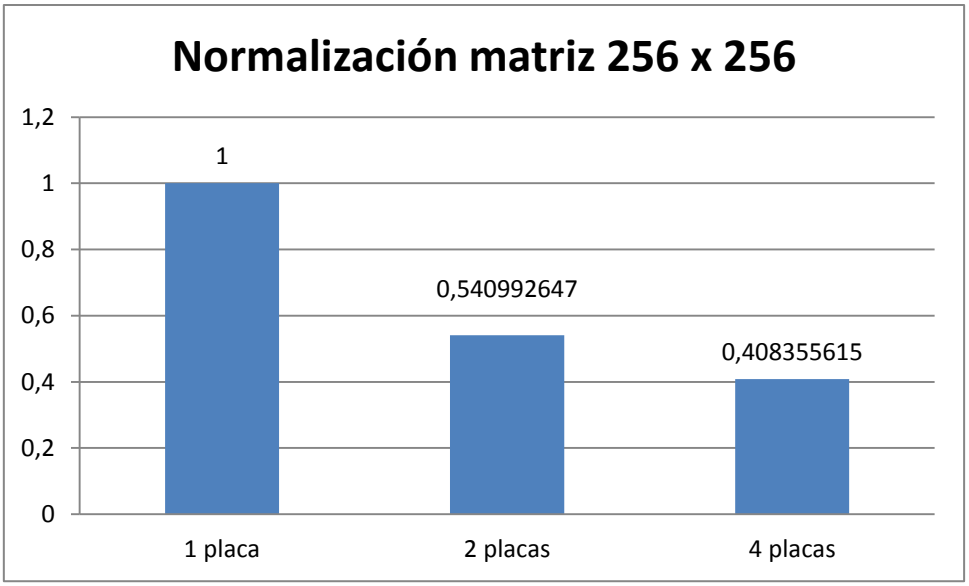


Ilustración 52.Normalizacion de resultados de la multiplicación de matrices 256 x 256.

Se puede destacar que al trabajar con 4 placas el beneficio obtenido con respecto a trabajar con una única placa es de 59.16 % mientras que al paralelizar trabajo con 2 placas se obtiene un beneficio del 45.90 %. Estos resultados sumados a los obtenidos por ***Sherihan Abu ElEnin y Mohamed Abu ElSoud*** apuntan que para resolver un problema de multiplicación de matrices con dimensiones 256 x 256 lo más eficiente es dividir el trabajo entre diferentes procesadores ya que la productividad se incrementa.

4.3.2 Matriz de tamaño 512 x 512

Al igual que en el sub apartado anterior, se expondrán los resultados obtenidos por ***Sherihan Abu ElEnin y Mohamed Abu*** para posteriormente ser comparados con los obtenidos en nuestra propuesta.

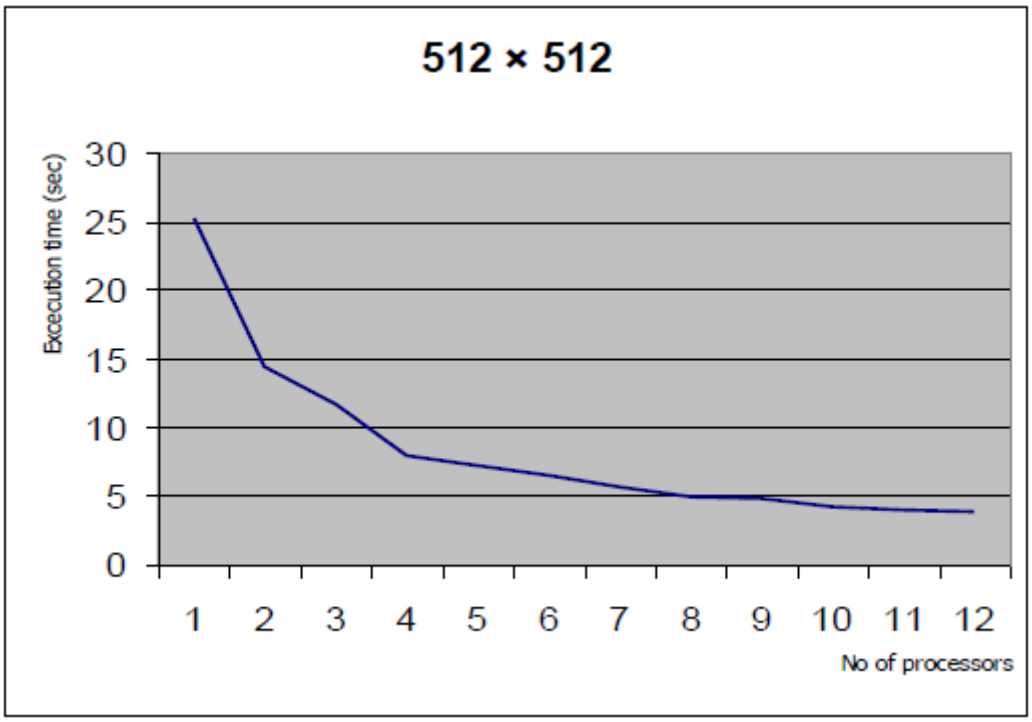


Ilustración 53. Resultados de la multiplicación de matrices 512 x 512 realizados por Sherihan Abu ElEnin y Mohamed Abu ElSoud [18].

Como se puede ver en la ilustración 53, la tendencia de los resultados obtenidos por la matriz 512 x 512 son parecidos a los mostrados por la matriz 256 x 256, para continuar con el análisis se procederá a presenta los resultados obtenidos por nuestra propuesta:

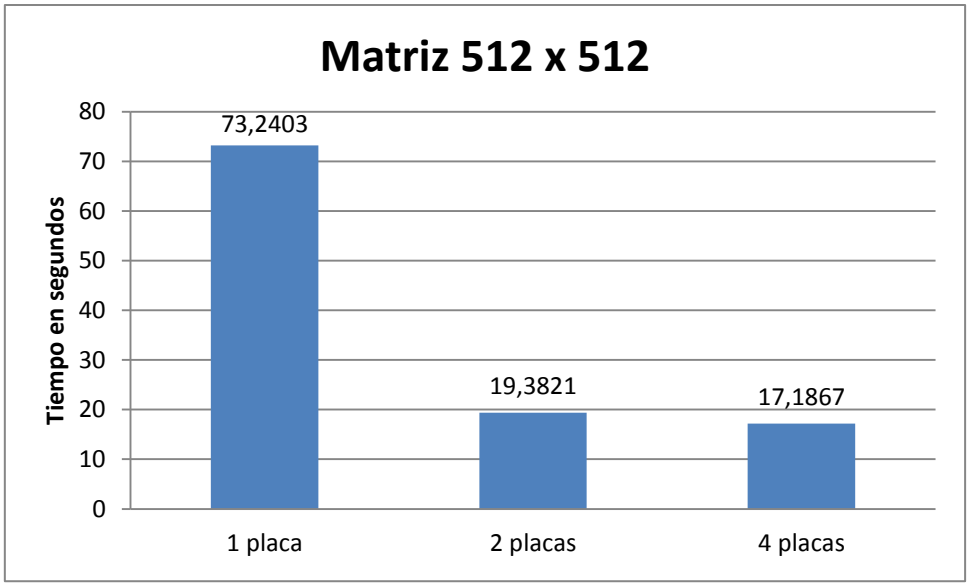


Ilustración 54. Resultados de la multiplicación de matrices 512 x 512.

Como se pudo observar en la ilustración 54, la tendencia es claramente decreciente debido a que este algoritmo hace un uso excesivo de CPU por lo que al repartir cálculos entre las diferentes placas permite acelerar el proceso y obtener el resultado de forma más rápida y eficiente; A continuación en la ilustración 55 se pueden apreciar los mismos resultados presentados en la ilustración 54 pero representados mediante una gráfica normalizada, en el caso de paralelizar trabajo con 4 placas se obtuvo un beneficio del 76,53 % mientras que trabajar con 2 placas fue de 73,53 %.

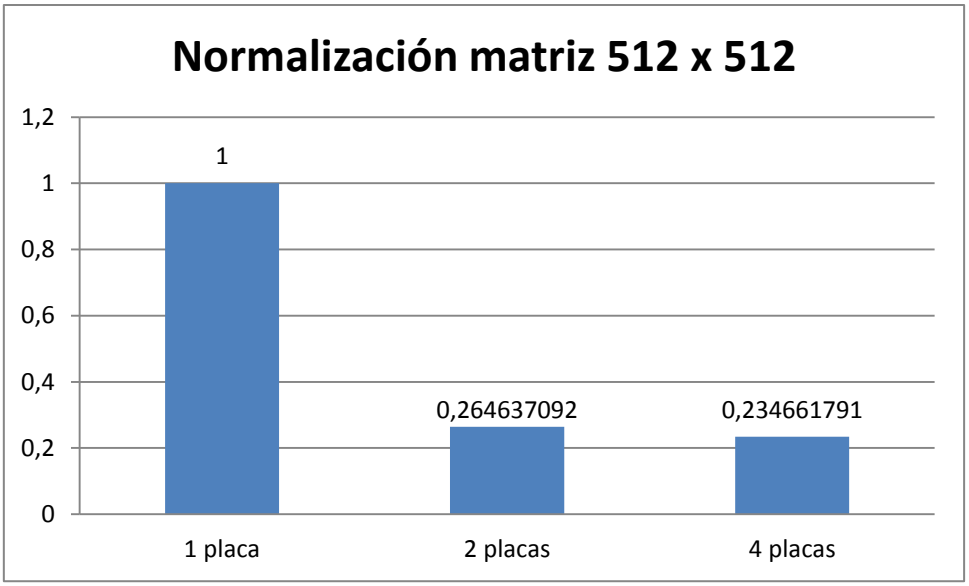


Ilustración 55. Normalización de resultados de la multiplicación de matrices 512 x 512.

4.3.3 Matriz de tamaño 1024 x 1024

A continuación se presentan los resultados obtenidos por ***Sherihan Abu ElEnin y Mohamed Abu*** para dicha matriz:

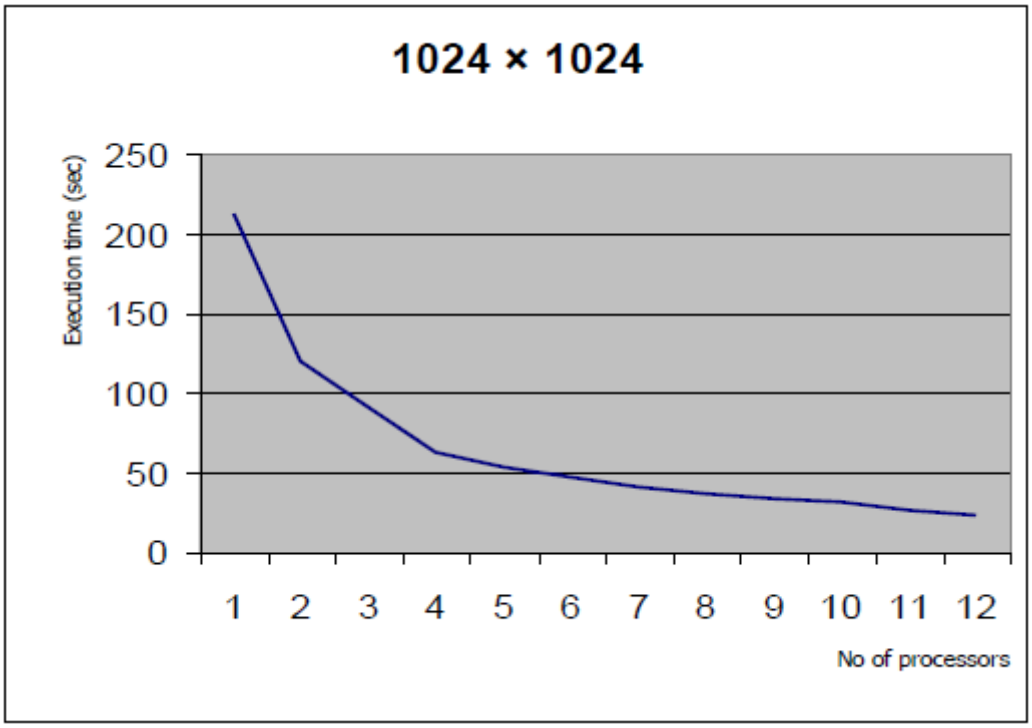


Ilustración 56 Resultados de la multiplicación de matrices 1024 x 1024 realizados por Sherihan Abu ElEnin y Mohamed Abu ElSoud [18].

Como en los sub-capítulos anteriores, se ilustraran los resultados obtenidos por nuestra propuesta para la matriz 1024 x 1024 para posteriormente ser analizados:

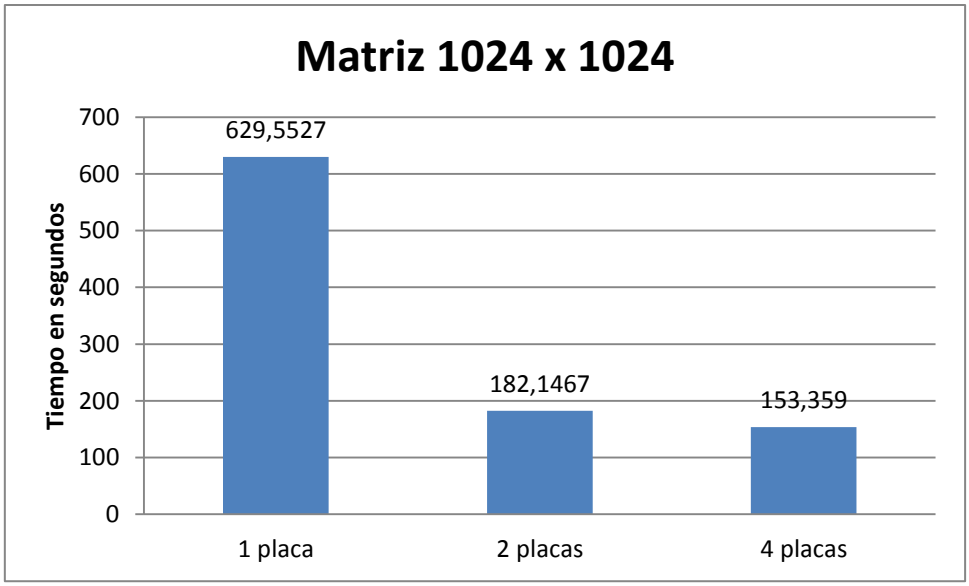


Ilustración 57 Resultados de la multiplicación de matrices 1024 x 1024.

La tendencia para esta matriz sigue siendo la igual a los resultados anteriores, a continuación se exponen los mismos resultados a través de una gráfica normalizada:

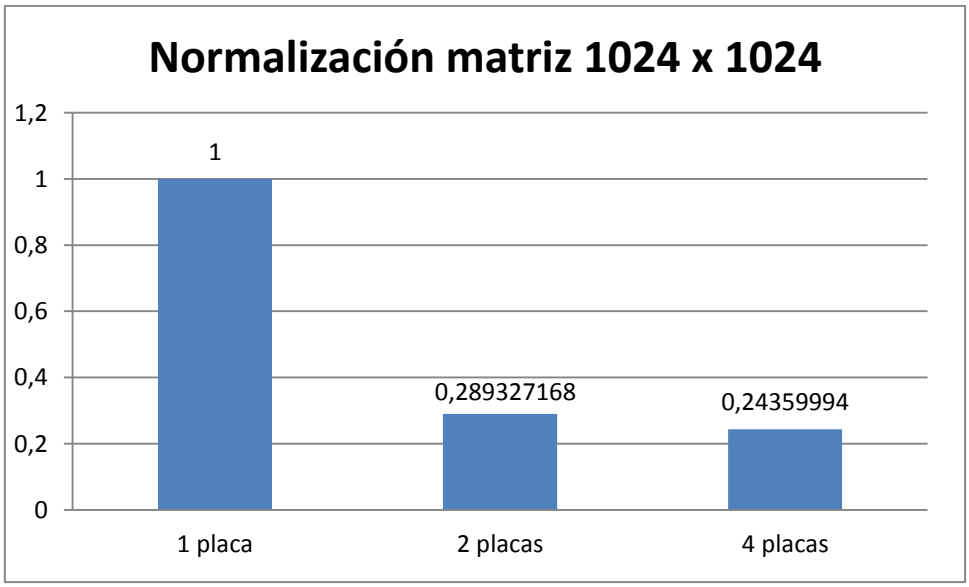


Ilustración 58 Normalización de resultados de la multiplicación de matrices 1024 x 1024.

Para estas dimensiones de la matriz al paralelizar con 4 placas se obtuvo un beneficio del 75.64 % mientras que para 2 placas se obtuvo del 71.06 %.

Para finalizar con el análisis del presente algoritmo, se expondrán los resultados obtenidos para una matriz con dimensiones de 2048 x 2048.

4.3.4 Matriz de tamaño 2048 x 2048

En el presente sub-capítulo, se terminarán de exponer los resultados obtenidos para el algoritmo de multiplicación de matrices, como se pudo observar en los sub-capítulos anteriores, las productividad se incrementa en relación a la cantidad de placas utilizadas (en nuestros experimentos solo se pudo paralelizar hasta 4 placas debido a que no disponíamos de más placas para distribuir trabajo). A continuación se exponen los resultados obtenidos por ***Sherihan Abu ElEnin y Mohamed Abu ElSoud:***

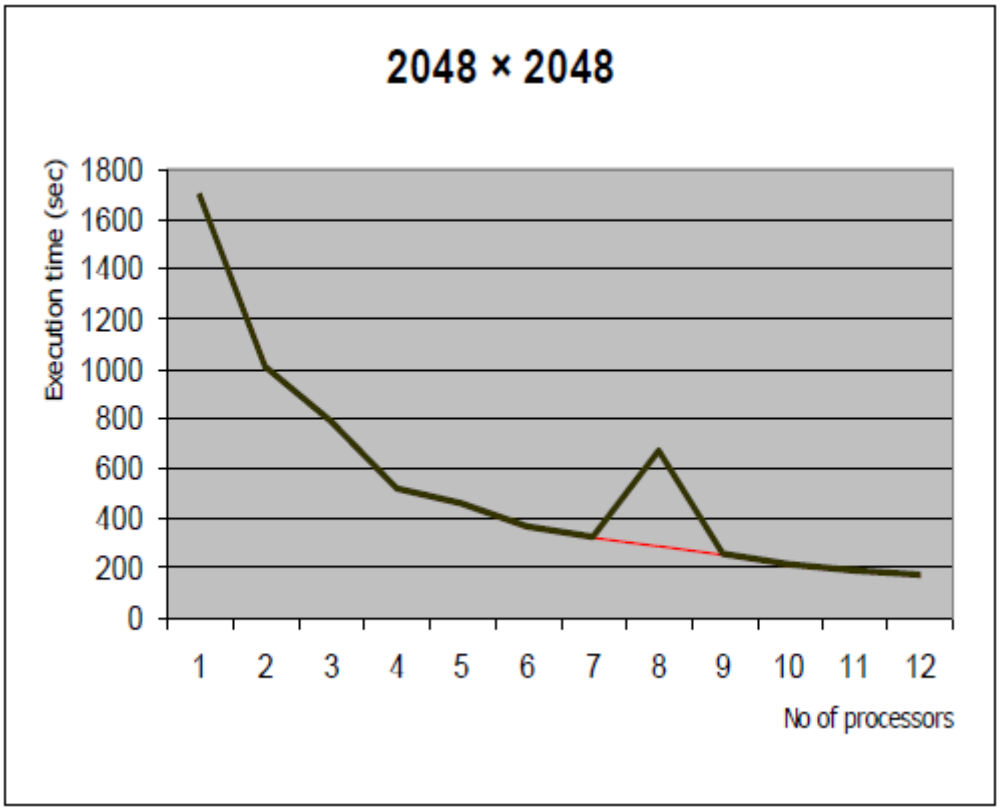


Ilustración 59 Resultados de la multiplicación de matrices 2048 x 2048 realizados por Sherihan Abu ElEnin y Mohamed Abu ElSoud [18].

A continuación se expondrán los resultados obtenidos en nuestra propuesta:

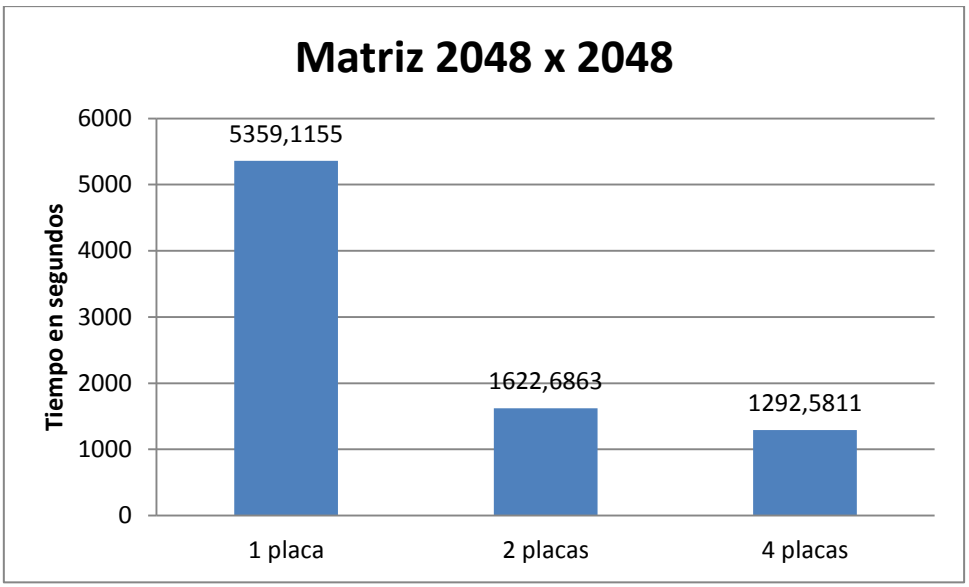


Ilustración 60 Resultados de la multiplicación de matrices 2048 x 2048.

Como se pudo apreciar en la ilustración 60, la tendencia es claramente descendiente como en los resultados expuestos anteriormente. Para finalizar se ilustraran los mismos resultados mediante una tabla normalizada.

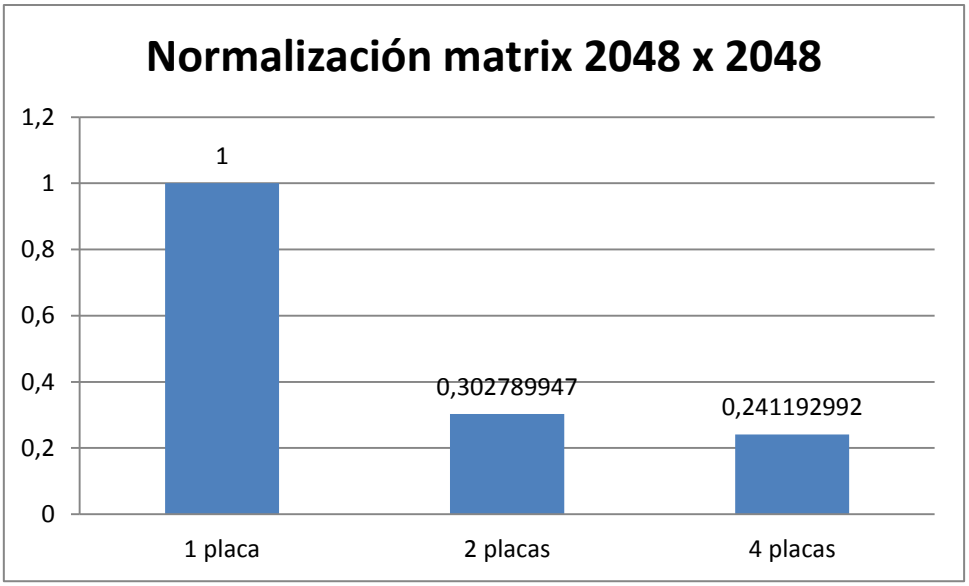


Ilustración 61 Normalización de resultados de la multiplicación de matrices 2048 x 2048.

El beneficio obtenido de trabajar con 4 placas fue del 75.88 % mientras que para 2 placas fue del 69.72 %. Para finalizar con el capítulo 4, a continuación se expondrán los resultados del último algoritmo realizado, el algoritmo de Montecarlo.

4.4 Resultados algoritmo Montecarlo para el cálculo de π

En el presente sub-capítulo se ilustrarán los resultados obtenidos mediante el algoritmo de Montecarlo, dichos resultados serán contrastados con otros proyectos de investigación para poder comparar la forma en que trabaja nuestra implementación. A continuación se exponen los resultados obtenidos por ***Y.Y. Teng & Z. G. HE*** [24] en su proyecto de investigación**.**

En sus resultados paralelizaban código entre diferentes cores utilizando OpenMP obteniendo los siguientes resultados:

Code	Runtime (seconds)	π value	Deviation
Code 1	34,6600	3,141368380	0,000224274
Code 2	14,2700	3,141634740	0,000042086
Code 3	5,5242	3,141548240	0,000054514
Code 4	0,9630	3,141507280	0,000029816

Ilustración 62. Resultados obtenidos por Y.Y. Teng & Z. G. HE [24].

Como se puede ver en la ilustración anterior, en sus experimentos entre más cores se utilicen con OpenMP mayor es la reducción en los tiempos de ejecución, pero esta reducción en el tiempo no significa que el número π sea más exacto ya que se consigue una mejor aproximación trabajando con tres cores en lugar de cuatro. Se podría concluir por sus resultados que la mejor aproximación posible de π es paralelizando hasta un total de tres cores.

Como en nuestra propuesta no es posible utilizar OpenMP (porque éste divide código entre los diferentes procesadores y la placa Intel Galileo Gen 2 solo posee un procesador y un único thread) se utilizara MPI entre las diferentes placas para poder comparar nuestros resultados a los obtenidos por ***Y.Y Teng & Z. G. HE.*** En la sección 3.4.2 se explicó cómo se crean procesos y como pueden ser distribuidos entre las diferentes placas.

A continuación en la ilustración 63 se puede apreciar la cantidad de iteraciones que realizaron en sus experimentos para poder estimar el valor de **π**, el tiempo invertido y el Speedup obtenido.

Iterations (million)	π value	Serial time	Parallel time	Speed up
200	3,14150728	2,549	0,963	2,647
2000	3,14162246	21,915	9,354	2,342

Ilustración 63 Recursos invertidos por Y.Y. Teng & Z. G. HE para calcular π [24].

En sus experimentos para poder estimar el valor de **π** hicieron falta 200 y 2000 millones de iteraciones (una iteración se puede definir como la cantidad de números generados aleatoriamente dentro del área definida en el capítulo 2.4.1), en nuestro caso para intentar simular sus resultados se optó por calcular los valores comprendidos entre 200 hasta 2000 (millones) iteraciones, A continuación en la ilustración 64 se puede ver de forma resumida los resultados obtenidos en nuestra propuesta:

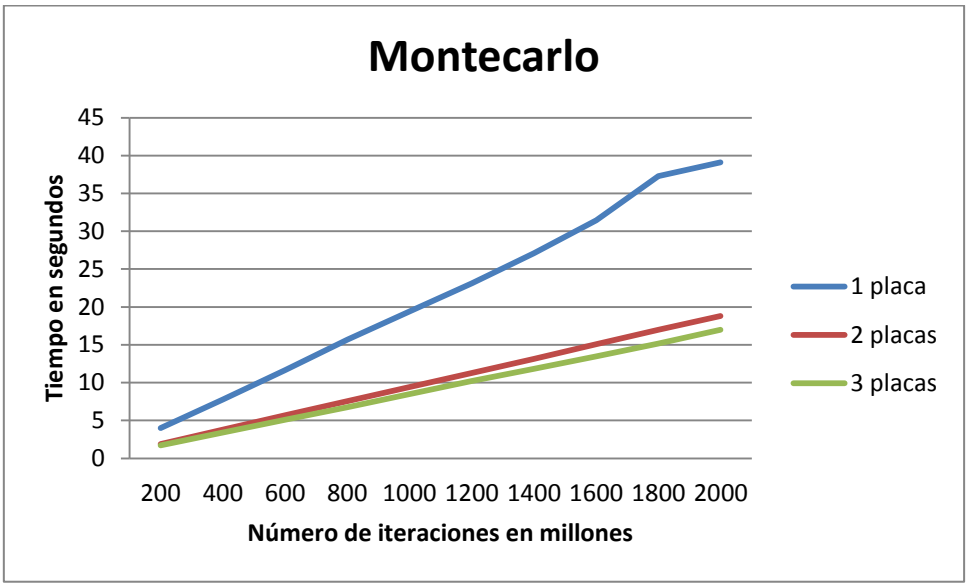


Ilustración 64 Tiempos para calcular π mediante el algoritmo de Montecarlo.

Como se pudo observar en la ilustración 64, el tiempo se reduce entre más se paralelice el problema, esto es debido a que al dividir el número de iteraciones k, la cantidad total de cálculos que una placa debe realizar se ve reducida significativamente. A continuación en la ilustración 65 se ilustran los mismos resultados pero para 2 y 3 placas.

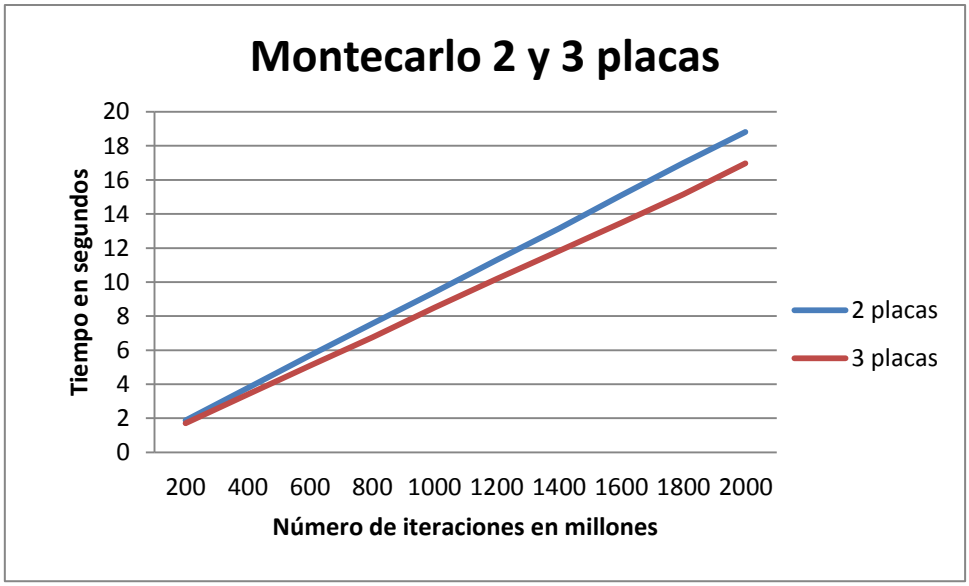


Ilustración 65 Tiempos para calcular π mediante el algoritmo de Montecarlo con 2 y 3 placas.

A continuación en la tabla 4 se exponen los resultados obtenidos en nuestra propuesta:

200 millones de iteraciones

	Tiempo de calculo	valor de π	desviación típica del valor de π
1 Placa	3,9933015	3,141600	0,00113
2 Placas	1,884494164	3,141576	0,01504
3 Placas	1,702138003	3,141540	0,04311

Tabla 5 Resultados obtenidos para 200 millones de iteraciones.

Los tiempos de cálculo se reducen entre más placas se utilicen, también la desviación típica es baja por lo que los resultados no varían en gran medida entre las diferentes ejecuciones. A continuación en la ilustración 66 se pueden observar los mismos resultados a través de una gráfica normalizada.

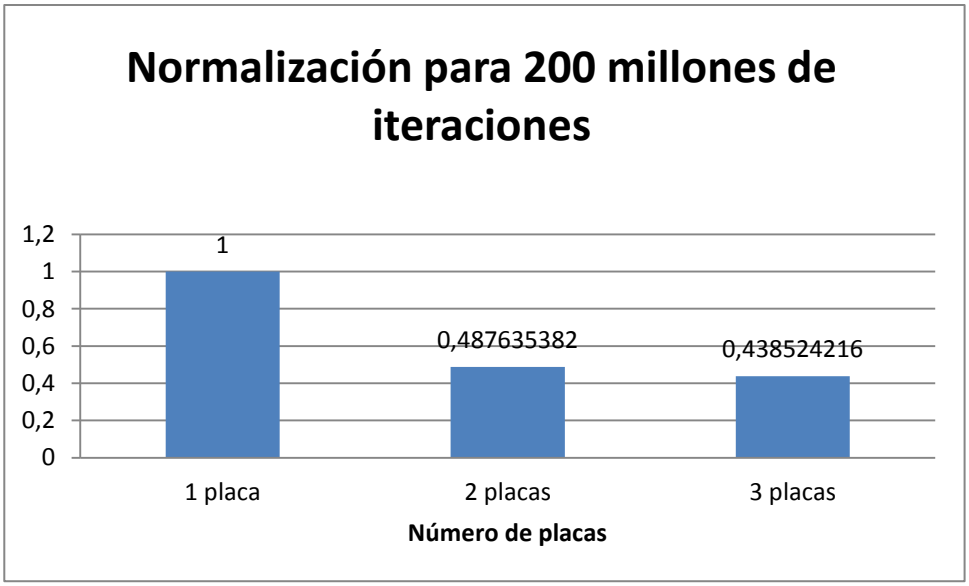


Ilustración 66 Normalización de tiempos de ejecución en algoritmo Montecarlo para 200 millones de iteraciones.

Al paralelizar el algoritmo entre 2 placas se obtuvo un beneficio del 51,23 % mientras que al paralelizarlo con 3 placas se obtiene un beneficio del 56,14%. Con estos resultados se terminan de exponer los resultados para el algoritmo de Montecarlo con 200 millones de iteraciones, a continuación se expondrán los resultados obtenidos para 2000 millones.

2000 millones de iteraciones

	Tiempo de calculo	valor de π	desviación típica del valor de π
1 Placa	39,1109225	3,141580	0,01204
2 Placas	18,81782276	3,141590	0,01210
3 Placas	16,98215881	3,141596	0,02567

Tabla 6 Resultados obtenidos para 2000 millones de iteraciones.

Como se exhibe en la tabla 4, el valor estimado de π es más aproximado con dos placas, a continuación se expondrán los mismos resultados mediante graficas normalizadas.

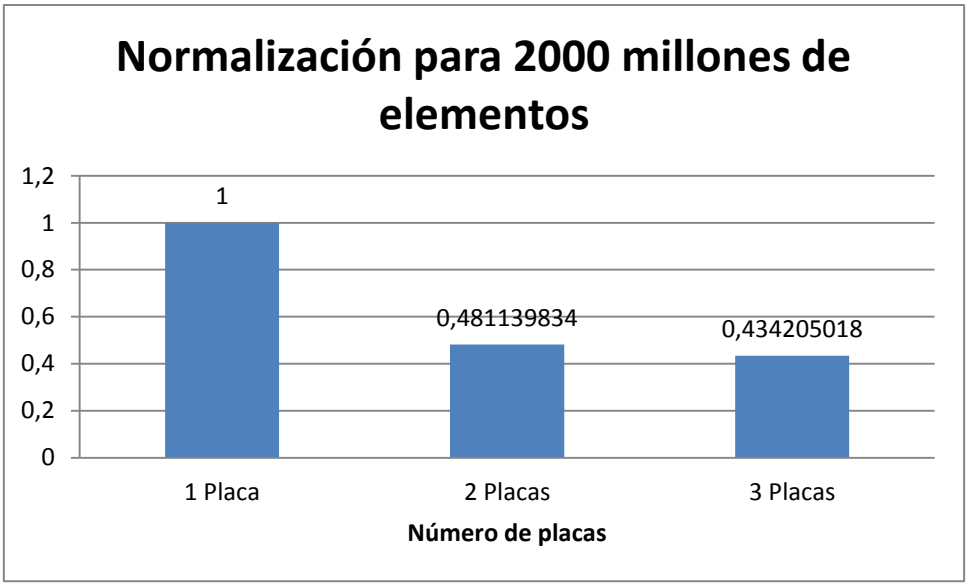


Ilustración 67 Normalización de tiempos de ejecución en algoritmo Montecarlo para 2000 millones de iteraciones.

Al paralelizar el algoritmo en 2 placas se obtuvo un beneficio del 51,88 % mientras que al paralelizarlo con 3 placas se obtuvo del 56,579%.

Cabe destacar que el speedup obtenido al comparar los resultados de dos a tres placas es mínimo debido a la forma en que fue implementado el algoritmo, se optó por utilizar un método de MPI denominado *MPI_Reduce* [25] que permite realizar operaciones matemáticas (en nuestro caso la adición) entre las diferentes placas, a continuación se puede ver un pequeño extracto del anexo J donde viene expuesto el algoritmo implementado para el cálculo de π mediante MPI:

```
/*
 *  numero de puntos dentro
 *  */
    temp = in;
    ierr = MPI_Reduce ( &temp, &numTotalesDentroSemiCirculo, 1, MPI_INT,
MPI_SUM, 0, workers );


/*
 *  numero de puntos fuera
 *  */
    temp = out;
    ierr = MPI_Reduce ( &temp, &numTotalesFueraSemiCirculo, 1, MPI_INT,
MPI_SUM, 0, workers );
```

Este método permite realizar la adición siempre y cuando las variables estén declaradas en todas las placas (en nuestra implementación se les denomino *numTotalesDentroSemiCirculo* y *numTotalesFueraSemiCirculo*). Una vez que la placa ha terminado con su proceso de cálculo (cualquier placa), esta se queda a la espera de que el resto de placas lleguen hasta la sentencia *MPI_Reduce* [25] para poder compartir los resultados obtenidos y poder realizar la adición.

Debido a la gran cantidad mensajes transmitidos por las diferentes placas, los tiempos de ejecución se vieron afectados ya que este método genera muchos retrasos no deseados.

Con todos los resultados ya expuestos, solo queda comparar los diferentes recursos hardware utilizados por cada algoritmo en las placas Intel Galileo Gen 2, dichas comparativas serán expuestas en el siguiente sub-capitulo.

4.5 Comparación y análisis de resultados obtenidos

En el presente sub-capítulo se terminarán de exponer los resultados obtenidos por los diferentes algoritmos, estos resultados serán ilustrados mediante gráfica en las cuales se exponen los diferentes requerimientos hardware (CPU y memoria RAM) invertidos por cada algoritmo para poder realizar sus cálculos. Cabe destacar que no hemos calculados los recursos utilizados por el algoritmo de Serie x^2 debido a que es un algoritmo con la única finalidad de comprender como es el funcionamiento de los programas ejecutados con MPI, además los resultados obtenidos serían muy similares a los utilizados por el algoritmo de Montecarlo.

4.5.1 Consumo de memoria RAM

En el presente sub-capitulo se expondrá el consumo de memoria ram invertido por los diferentes algoritmos (los resultados que se exponen a continuación fueron obtenidos exclusivamente por la placa principal). Para comenzar se expone el consumo de memoria ram utilizado por el sistema operativo, de esta forma se tendrá un punto de referencia para poder comparar los diferentes algoritmos:

```
[root@galileo-Master ~]# free -h
             total       used       free     shared    buffers     cached
Mem:          237M        51M       186M       196K       804K     16064K
-/+ buffers/cache:         44M       192M
[root@galileo-Master ~]#
```

Ilustración 68 Consumo de memoria RAM por el sistema operativo.

La placa Intel Galileo Gen 2 ejecutando exclusivamente el sistema operativo consume entre los 50 y 51 MB en memoria (los resultados de consumo de memoria ram fueron obtenidos por el comando *free* de debian [5]). A continuación se ilustrara de forma gráfica los resultados obtenidos por el algoritmo de quicksort:

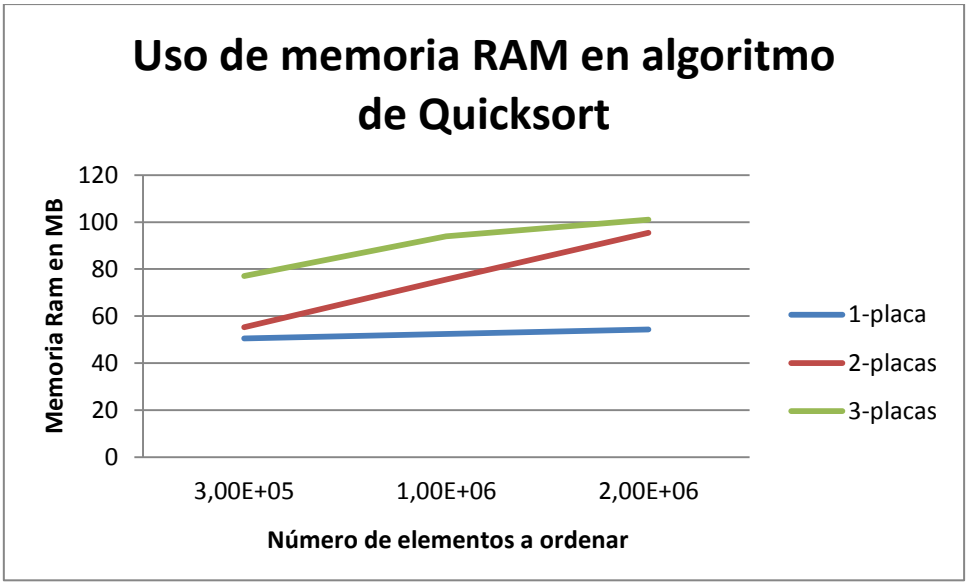


Ilustración 69 Consumo de memoria RAM en algoritmo Quicksort.

Como se puede apreciar en la ilustración 69, a medida que se aumenta el número de elementos a ordenar, el algoritmo quicksort programado en MPI requiere cada vez más de memoria ram (según la cantidad de placas a utilizar) para poder realizar sus cálculos, esto es debido a que MPI necesita mantener muchos recursos en memoria, como por ejemplo mantener las sesiones *ssh* activas, transferir y recibir los datos, etc.

Por otra parte, el algoritmo quicksort programado de forma secuencial no requiere de tanta memoria ram debido a que esté solo necesita gestionar la memoria reservada para el array. A continuación se expone los resultados obtenidos por el algoritmo de multiplicación de matrices.

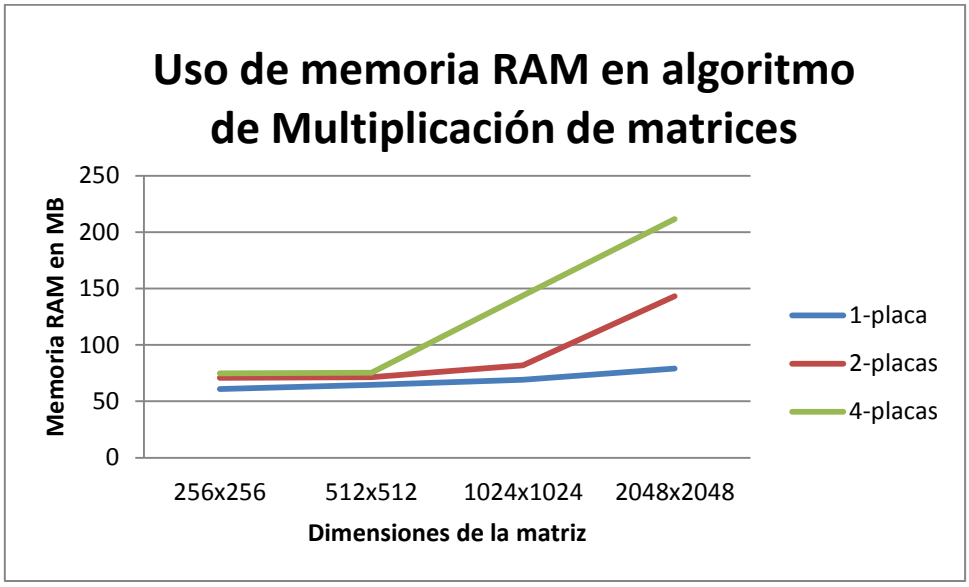


Ilustración 70 Consumo de memoria RAM en algoritmo Multiplicación de matrices.

Los resultados expuestos en la ilustración 70 son muy similares a los resultados obtenidos por el algoritmo quicksort, a pesar de que MPI mejora los tiempos de ejecución, éste requiere de una gran cantidad de memoria ram. También se puede concluir que la memoria ram consumida por las matrices 256x256 y 512x512 es constante por las librerías cargadas. Para finalizar se expondrán los recursos utilizados por el algoritmo de Montecarlo

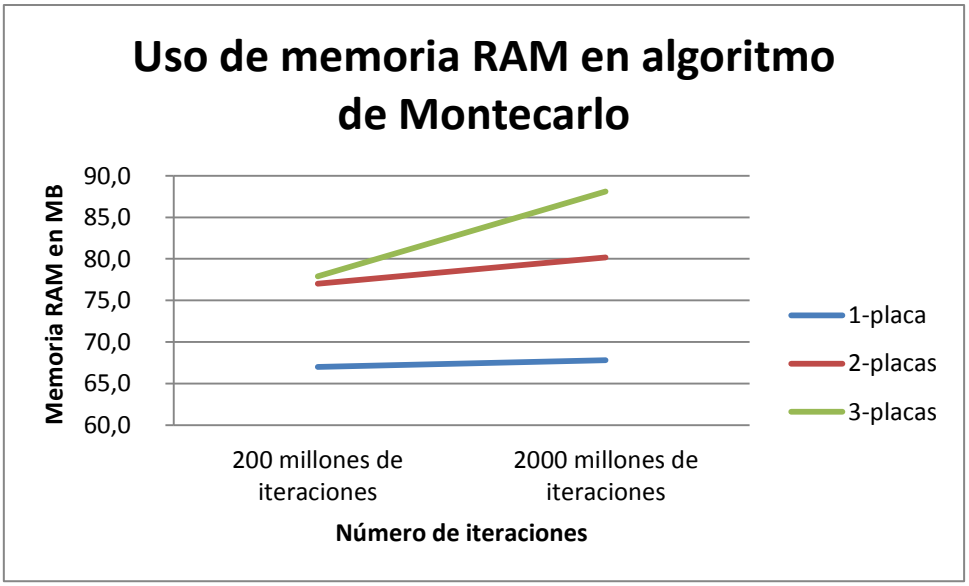


Ilustración 71 Consumo de memoria RAM en algoritmo Montecarlo.

Con los resultados obtenidos en la ilustración 71, se podría concluir que MPI es un estándar excelente cuando la finalidad del programa es solucionar un problema de la forma más rápida posible, pero se debe de tener en cuenta que dicho estándar requiere de grandes cantidad de recursos por lo que si el programa necesita grandes cantidades de memoria (mayor a la cantidad de memoria ram disponible) no sería muy eficiente ya que éste terminaría utilizando memoria virtual generando retardos no deseados.

4.5.2 Consumo de CPU

En el presente sub-capítulo se expondrá el porcentaje de utilización de la CPU por parte de los diferentes algoritmos implementados, tal y como se hizo en el sub-capítulo anterior, primero se expondrá el uso de la CPU únicamente cuando la placa Intel Galileo Gen 2 está ejecutando el sistema operativo. Adicionalmente, se debe de tener en cuenta que los resultados que se exponen a continuación son obtenidos exclusivamente por la placa principal.

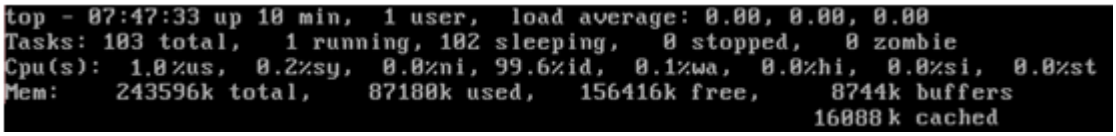


Ilustración 72 Consumo de CPU por el sistema operativo.

Como se pudo apreciar en la ilustración 72, el consumo de la CPU cuando únicamente se está ejecutando el sistema operativo es del uno por ciento, esta medida se debe de tomar como referencia para los resultados que serán expuestos a continuación (los resultados fueron obtenidos mediante el comando top de debían [5]). En la siguiente ilustración se expone el consumo de la CPU por el algoritmo quicksort.

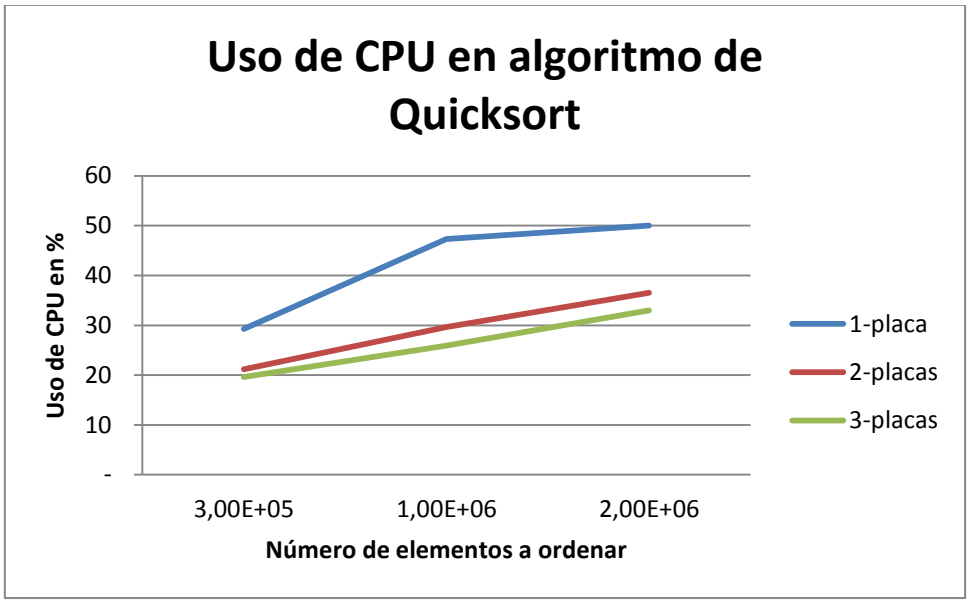


Ilustración 73 Consumo de CPU en algoritmo Quicksort.

Como se pudo observar en la ilustración 73, el consumo de cpu se reduce considerablemente en los algoritmos ejecutados mediante MPI, esto es debido a que MPI reserva la mayor cantidad de recursos posibles (como bien se pudo apreciar en los consumo de memoria en el sub-capítulo anterior) del sistema con la finalidad de terminar con la programa lo antes posible, viéndose esto reflejado en el consumo por parte de la cpu. En la siguiente ilustración se puede observar los resultados obtenidos al medir el consumo de cpu en el algoritmo de multiplicación de matrices.

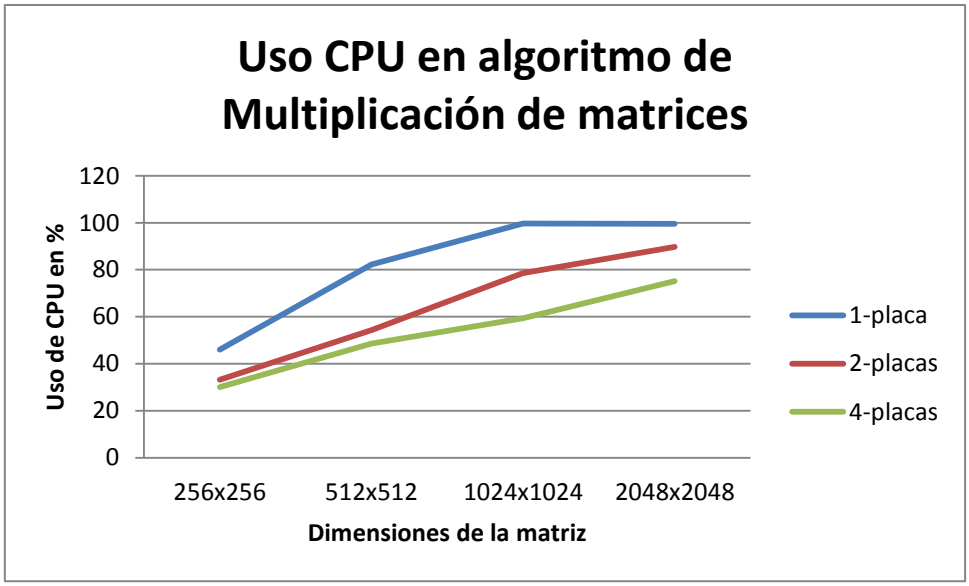


Ilustración 74 Consumo de CPU en algoritmo Multiplicación de Matrices.

Al observar los resultados obtenidos por la ilustración 74, se puede apreciar como el uso de la CPU se incrementa de forma muy drástica al ejecutar el algoritmo en una única placa, tan drástico es el aumento que satura completamente el sistema y no es posible obtener medidas a través de conexiones *ssh*, por este motivo, nos vimos obligados a obtener los resultados mediante un script el cual nos informaba del consumo del algoritmo al finalizar su ejecución. En la siguiente ilustración se puede apreciar el consumo de CPU para una matriz de tamaño 2048 x 2048.

```
Tasks: 202 total, 101 running, 101 sleeping,   0 stopped,   0 zombie
Cpu(s): 99.8%us,  0.1%sy,  0.0%ni,  0.0%id,  0.0%wa,  2.0%hi,  1.5%si,  0.0%st
Mem:    243596k total,   239652k used,     3944k free,      108k buffers
```

Ilustración 75 Consumo de CPU en algoritmo Multiplicación de Matrices para una matriz de dimensiones 2048 x 2048.

Como se pudo apreciar, el sistema se satura completamente debido a la gran cantidad de cálculos que una placa debe de realizar, es por ese motivo que al tener un algoritmo con este nivel de desgaste de recursos merece la pena utilizar MPI en lugar de la programación secuencial.

Para finalizar, se expondrá el uso de CPU por el último algoritmo realizado.

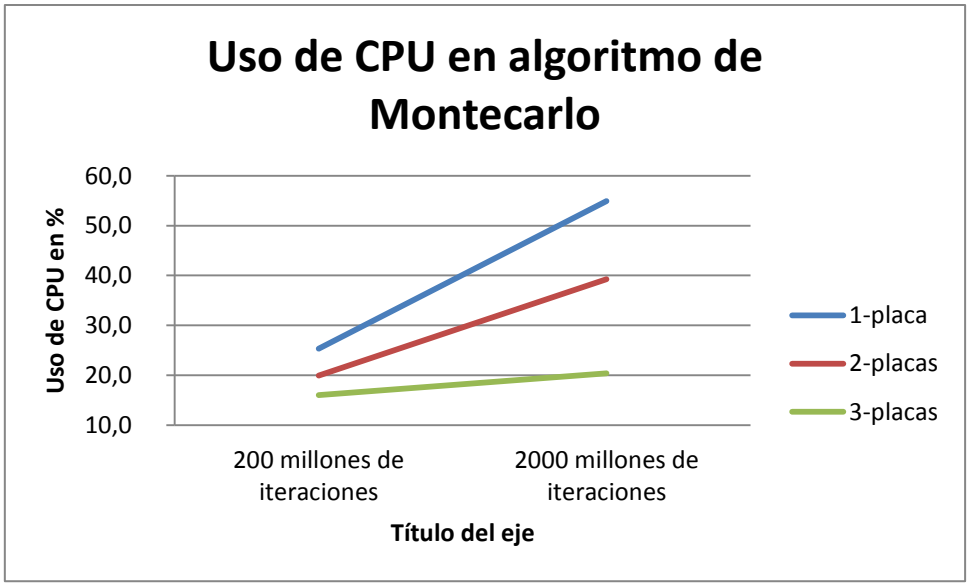


Ilustración 76 Consumo de CPU en algoritmo Montecarlo.

Al observar el consumo de la CPU, éste es más elevado cuando el algoritmo se ejecuta con una única placa, a medida que se divide más la carga de trabajo el consumo de la CPU disminuye. Se podría concluir que gracias a un aumento del consumo en la memoria ram MPI le permite al sistema disminuir la cantidad de cálculos que la CPU debe de realizar (mejorando también los tiempos de ejecución). Con estos resultados se concluye el capítulo 4. A continuación se expondrá el capítulo de resultados en el cual se expondrán una serie de conclusiones de los experimentos realizados en el presente proyecto de investigación.

Capítulo 5. Conclusiones

El objetivo de este proyecto fin de grado fue desarrollar diferentes algoritmos (tanto en secuencial como en paralelo) para un grid de sistemas empotrados con la finalidad de paralelizar la carga de trabajo entre las diferentes placas Intel Galileo Gen 2. Tal y como se pudo observar a lo largo del capítulo 4, dicho objetivo fueron cumplidos de forma eficiente ya que el rendimiento obtenido por los programas implementados de forma paralela son más eficaces a los resultados obtenidos por una única placa.

Gracias a este ahorro en tiempos de ejecución, en el grid implementado se podría resolver casi cualquier problema u algoritmo (siempre y cuando éste sea paralelizable) con la finalidad de ahorrar el recurso más importante que existe, el tiempo; entre más tiempo se consigue ahorrar más problemas se pueden intentar resolver por lo que es altamente recomendable utilizar MPI.

Adicionalmente se observa que para algoritmos más sencillos, como por ejemplo el algoritmo de serie x^2, el rendimiento de MPI depende mucho de la cantidad de adiciones se desea realizar, ya que si éste es bajo los tiempos de ejecución resultantes serán muy superiores a los obtenidos por una única placa. Por el contrario, sí el algoritmos requiere de un mayor consumo de memoria RAM o CPU, observamos que MPI genera un mayor beneficio gracias a la distribución de trabajo entre las diferentes placas.

Cabe destacar que MPI realiza un uso excesivo de la memoria RAM (como se demostró a lo largo del sub-capítulo 4.5.1) por lo que si el algoritmo a desarrollar necesita de grandes cantidades de memoria (más grandes que la cantidad de memoria RAM disponibles) no sería muy aconsejable utilizarlo (ya que el sistema se vería forzado a utilizar memoria virtual generando retardos no deseados). Sin embargo, si el sistema dispusiera de grandes cantidades de memoria, sería aconsejable paralelizar el algoritmo (si es posible) con MPI ya que éste reduce el consumo de la CPU y los tiempos de ejecución.

Para finalizar se puede concluir que al utilizar una interfaz como MPI, junto a algoritmos bien conocidos y ejecutados sobre un hardware limitado, es posible

paralelizar la carga de trabajo sin necesidad de ningún software adicional más que las librerías instaladas por defecto en el sistema operativo debian, Creemos que esta reducción de costes es esencial para hacer más atractivo el estudio y comercialización de proyectos que se basen en la distribución de trabajo entre los diferentes procesadores y placas.

Chapter 5 Conclusions

The objective of this project was to develop different algorithms for a grid of embedded systems in order to parallelize the workload between the boards Intel Galileo Gen 2, as was observed throughout chapter 4, this objectives was accomplished as the results obtained by the programs implemented in parallel are better to those obtained by a single board.

Thanks to this saving the developed grid could try to resolve almost any problem or algorithms (as long as it's parallelizable) in order to save the most important resources that exist, the time; that's why MPI is the best choice because it's focus on reduce the executions times of the program.

It's also observed that for a simpler algorithm, such as the Series x^2, MPI performance depends greatly on the number of additions to be made, since if it's under the resulting execution times, it will be much higher than those by a single board. Conversely, if the algorithms require more RAM memory or CPU, we note that MPI generate a greater benefit from the distribution of workload between the boards.

One remarkable thing is that MPI makes an excessive use of memory RAM (as demonstrated throughout chapter 4) so if the algorithm to be develop requires a large amount of memory (larger than the amount of memory RAM available) use MPI would not be advisable option (because the system will be force to use virtual memory generating unwanted delays). However, if the system had a large amount of RAM memory, the use of MPI would be advisable with the finality to parallelize the algorithm (if it's possible) and reduce the use of CPU and execution times.

Finally can be concluded that when using an interface like MPI, along with different algorithms and executed on a limited hardware, it's possible to parallelize the workload without the needing of any additional software more than the libraries installed by default in the operative system. We believe that this cost reduced is essential to make more attractive the study and commercialization of project that are based on the distributed of work between the different processors and boards.

Capítulo 6 Líneas Futuras

El proyecto ha consistido en buena parte en la realización de una investigación profunda y continúa sobre la placa Intel Galileo Gen 2, ya que ésta es relativamente nueva, poco conocida y muy restrictiva por su hardware limitado. Sin embargo, a lo largo del curso han surgido ideas y modificaciones a realizar que no eran factibles en este periodo de tiempo.

Pero a pesar de los inconvenientes, el grid desarrollado permitió a los diferentes algoritmos programados de forma paralela obtener mejores tiempos de ejecución que los programados de forma secuencial.

A continuación se detalla una serie de sugerencias para futuros proyectos de investigación:

- Cambiar la frecuencia del procesador con la finalidad de analizar los nuevos resultados para diferentes algoritmos.
- Realizar un estudio sobre el consumo energético para diferentes algoritmos programados de forma secuencial y paralela.
- Realizar un nuevo estudio comparativo para una nueva generación de hardware.

Anexo A. Serie x^2 Secuencial

```
# include <stdio.h>
# include <stdlib.h>
# include <time.h>

unsigned long long int calculoSerie (unsigned long long int n){
        unsigned long long int i, total=0, cuadrado=0;
        for(i=0;i<n+1;i++){
                cuadrado=i*i;
                total= total + cuadrado;
        }
        return total;
}

int main ( int argc, char *argv[] ){
        unsigned long long int result=0;
        unsigned long long int numCalculo =
strtoul(argv[1],NULL,10);
        result=calculoSerie(numCalculo);


        printf("N:%i\n",numCalculo);
        printf("%i\n",result);
        printf("MEDIA:%6.3f\n",aux);

        return 0;
}
```

Anexo B. Serie x^2 MPI con 2 nodos

```
# include <stdio.h>
# include <stdlib.h>
# include <time.h>
# include "/usr/include/mpi/mpi.h"
#include "math.h"
#include <sys/time.h>


int main ( int argc, char *argv[] ){

# define N 1000

  unsigned long long int array[N];
  int i, puntoMedio = 0;
  int master = 0;
  int my_id;
  int numprocs;
  MPI_Status status;
  double sum;
  double sum_all;
  double aux = 0;
  unsigned long long int sumaPar = 0;
  unsigned long long int sumaImpar = 0;
  unsigned long long int sumaTotal = 0;
  unsigned long long int arrayVuelta[N];

/*
  Initialize MPI.
*/
  MPI_Init ( &argc, &argv );
/*
  Get the number of processes.
*/
  MPI_Comm_size ( MPI_COMM_WORLD, &numprocs );
/*
  Determine the rank of this process.
*/
  MPI_Comm_rank ( MPI_COMM_WORLD, &my_id );

/*
  The master process initializes the array.
*/

  unsigned long long int numCalculo = strtoul(argv[1],NULL,10);
  array[0] = numCalculo;
  puntoMedio = numCalculo / 2;

  MPI_Bcast ( array, N, MPI_LONG_LONG, master, MPI_COMM_WORLD );

  if ( my_id == master ){
       unsigned long long int i;
       for ( i = 0; i < puntoMedio; i++ ){

                        sumaPar = sumaPar + (i*i);
```

```
        }
    }
    else{
        unsigned long long int j;
         for ( j = puntoMedio; j < array[0] + 1; j++ ){

                    sumaImpar = sumaImpar + (j*j);

         }
        arrayVuelta[0] = sumaImpar;
    }
    if ( my_id != master ){
        MPI_Send ( arrayVuelta, N, MPI_INT, master, 0,
MPI_COMM_WORLD );
    }
    else{
        MPI_Recv ( arrayVuelta, N, MPI_INT, 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE );
    }

    if ( my_id == master){
        sumaTotal = sumaPar + arrayVuelta[0];
        printf("N:%s\n",argv[1]);
        printf("TOTAL:%i\n",sumaTotal);
        //printf("TOTAL:%i\n",numCalculo);
        //printf("TOTAL:%i\n",puntoMedio);
    }

    MPI_Finalize ( );

    return 0;
# undef N
}
```

Anexo C Serie x^2 MPI con 3 nodos

```
# include <stdio.h>
# include <stdlib.h>
# include <time.h>
# include "/usr/include/mpi/mpi.h"
#include "math.h"
#include <sys/time.h>

int main ( int argc, char *argv[] ){

# define N 1

  unsigned long long int array[N];
  int master = 0;
  int my_id;
  int numprocs;
  MPI_Status status;

  unsigned long long int sumaTotal = 0;
  unsigned long long int suma1 = 0;
  unsigned long long int suma2 = 0;
  unsigned long long int suma3 = 0;
  unsigned long long int arrayVuelta_1[N];
  unsigned long long int arrayVuelta_2[N];
/*
  Initialize MPI.
*/
  MPI_Init ( &argc, &argv );
/*
  Get the number of processes.
*/
  MPI_Comm_size ( MPI_COMM_WORLD, &numprocs );
/*
  Determine the rank of this process.
*/
  MPI_Comm_rank ( MPI_COMM_WORLD, &my_id );

/*
  The master process initializes the array.
*/

  unsigned long long int numCalculo = strtoul(argv[1],NULL,10);
  array[0] = numCalculo;
  unsigned long long int tercio = numCalculo / 3;

  MPI_Bcast ( array, N, MPI_LONG_LONG, master, MPI_COMM_WORLD );

  if ( my_id == master ){
        unsigned long long int i;
        for ( i = 0; i < tercio; i++ ){
                suma1 = suma1 + (i*i);
        }
  }
  else{
        if( my_id == 1 ){
                unsigned long long int j;
```

```
                for ( j = tercio; j <  tercio*2 ; j++ ){
                            suma2 = suma2 + (j*j);
                }
                arrayVuelta_1[0] = suma2;
        }
        else{
                unsigned long long int k;
                 for ( k = tercio*2; k <= numCalculo; k++ ){
                            suma3 = suma3 + (k*k);
                 }
                 arrayVuelta_2[0] = suma3;
        }
  }
  if ( my_id != master ){
        if( my_id == 1 ){
                MPI_Send ( arrayVuelta_1, N, MPI_INT, master, 0,
MPI_COMM_WORLD );
        }
        else{
                MPI_Send ( arrayVuelta_2, N, MPI_INT, master, 0,
MPI_COMM_WORLD );
        }
  }
  else{
       MPI_Recv ( arrayVuelta_1, N, MPI_INT, 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE );
       MPI_Recv ( arrayVuelta_2, N, MPI_INT, 2, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE );
  }

  if ( my_id == master){
        sumaTotal = suma1 + arrayVuelta_1[0] + arrayVuelta_2[0];
        printf("N:%s\n",argv[1]);
        printf("TOTAL:%i\n",sumaTotal);
        //printf("TOTAL:%i\n",numCalculo);
        //printf("TOTAL:%i\n",puntoMedio);
  }

  MPI_Finalize ( );

  return 0;
# undef N
}
```

Anexo D algoritmo Quicksort secuencial

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define N 1000000

void quickSort( int a[], int l, int r);
int pivote( int a[], int l, int r);
void imprimir( int a[], int max);

void main()
{
	int i = 0, j = 0;

	int *a;

	a = (int*)malloc(N*sizeof(int));

	srand (time(NULL));

	for(i=0;i<N-1;i++)
	     a[i] = random();

	clock_t start = clock();
	quickSort( a, 0, N - 1);
	double aux = ((double)clock() - start) / CLOCKS_PER_SEC;;

	printf(" %f",aux,"\n");
	printf("\n");

	imprimir(a,N);
}

void imprimir( int a[], int max){
	for (int i = 0; i < max-1; i++)
		printf(" %i",a[i],"\n");
}


void quickSort( int a[], int l, int r)
{
   int j;

   if( l < r )
   {
       j = pivote( a, l, r);
       quickSort( a, l, j-1);
       quickSort( a, j+1, r);
   }
}

int pivote( int a[], int l, int r) {
   int pivot, i, j, t;
   pivot = a[l];
   i = l; j = r+1;
```

```
    while(1)
    {
        do ++i; while( a[i] <= pivot && i <= r );
        do --j; while( a[j] > pivot );
        if( i >= j ) break;
        t = a[i]; a[i] = a[j]; a[j] = t;
    }
    t = a[l]; a[l] = a[j]; a[j] = t;
    return j;
}
```

Anexo E algoritmo Quicksort MPI con 2 placas

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include </usr/include/openmpi-x86_64/mpi.h>

#define N 1000000

void quickSort( int a[], int l, int r);
int partition( int a[], int l, int r);

int main(int argc, char *argv[])
{
        int master = 0;
         int my_id;
         int numprocs;
         MPI_Status status;

        /* se inicializa las variables de MPI, se obtiene el numero
de placas a utilizar y obtenemos el process id de la placa
principal */

        MPI_Init ( &argc, &argv );
        MPI_Comm_size ( MPI_COMM_WORLD, &numprocs );
        MPI_Comm_rank ( MPI_COMM_WORLD, &my_id );

        srand (time(NULL));

        int *b,*c;

        /*se generan los arrays b y c*/

        b = (int*)malloc((N/2)*sizeof(int));
        c = (int*)malloc((N/2)*sizeof(int));

        int i = 0, j = 0;

        /*se generan los numeros aleatorios en los arrays*/

        if( my_id == master) {
                for(i=0;i<N/2;i++){
                        b[i] = rand();
                        c[i] = rand();
                }
        }

        clock_t start = clock() ;

        /*se envia el array c a la placa esclava 1*/

        if ( my_id == 0 ){
                MPI_Send ( c, N/2, MPI_INT, 1, 0, MPI_COMM_WORLD );
        }
        else{
                MPI_Recv ( c, N/2, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE );
```

```
	}

	/*se ordenan los sub array por las diferentes placas*/

	if (  my_id == 0 ) {
		quickSort( b, 0, N/2);
	}
	else{
		quickSort( c, 0, N/2);
	}

	/*se obtiene los resultados por parte de la placa esclava
1*/

	if ( my_id == 1 ){
		 MPI_Send ( c, N/2, MPI_INT, 0, 0, MPI_COMM_WORLD );
	 }
	 else{
		 MPI_Recv ( c, N/2, MPI_INT, 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE );
	 }

	/*funcion merge que imprime los resultados*/

	if( my_id == 0  ) {

		int MAX_B = 0, MAX_C = 0, i = 0;

		for ( i = 0; i < N; i++) {
			if( MAX_B < N/2 ) {
				if( MAX_C < N/2 ){
					if (b[MAX_B] <= c[MAX_C]){
						printf(" %d ",
b[MAX_B]);
						MAX_B++;
					}
					else{
						printf(" %d ",
c[MAX_C]);
						MAX_C++;
					}
				}
				else{
					printf(" %d ", b[MAX_B]);
					    MAX_B++;
				}

			}
			else{
				printf(" %d ", c[MAX_C]);
				MAX_C++;
			}
		}

	}


	double aux = ((double)clock() - start) / CLOCKS_PER_SEC;
```

```
        /* se imprime el tiempo invertido en ordenar N elementos */

        if ( my_id == 0  ){
                printf("\n");
                printf("Tiempo de ejecucion quicksort en MPI con 2
placas: %f ", aux,"\n");
                printf("\n");
        }

        MPI_Finalize();
        return 0;
}

void quickSort( int a[], int l, int r)
{
   int j;

   if( l < r )
   {
       j = partition( a, l, r);
       quickSort( a, l, j-1);
       quickSort( a, j+1, r);
   }
}

int partition( int a[], int l, int r) {
   int pivot, i, j, t;
   pivot = a[l];
   i = l; j = r+1;

   while(1)
   {
       do ++i; while( a[i] <= pivot && i <= r );
       do --j; while( a[j] > pivot );
       if( i >= j ) break;
       t = a[i]; a[i] = a[j]; a[j] = t;
   }
   t = a[l]; a[l] = a[j]; a[j] = t;
   return j;
}
```

Anexo F algoritmo Quicksort MPI con 3 placas

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include </usr/include/openmpi-x86_64/mpi.h>

#define N 10

void quickSort( int a[], int l, int r);
int partition( int a[], int l, int r);

int main(int argc, char *argv[])
{
        int master = 0;
         int my_id;
         int numprocs;
         MPI_Status status;

        /*se inicializa MPI, se obtiene el id de la placa principal
y el numero total de procesos*/

        MPI_Init ( &argc, &argv );
        MPI_Comm_size ( MPI_COMM_WORLD, &numprocs );
        MPI_Comm_rank ( MPI_COMM_WORLD, &my_id );

        srand (time(NULL));

        /*se reservan los array de elementos a ordenar*/

        int *b,*c,*d;

        b = (int*)malloc((N/3)*sizeof(int));
        c = (int*)malloc((N/3)*sizeof(int));
        d = (int*)malloc((N/3)*sizeof(int));

        int i = 0, j = 0;

        /*se inicializan los arrays con números aleatorios*/

        if( my_id == master) {
                for(i=0;i<N/3;i++){
                        b[i] = rand();
                        c[i] = rand();
                        d[i] = rand();
                }
        }

        /* se comienza a medir el tiempo de ejecucion*/

        clock_t start = clock() ;

        /* se enviar los array de datos */
        if ( my_id == 0 ){
                MPI_Send ( c, N/3, MPI_INT, 1, 0, MPI_COMM_WORLD );
                MPI_Send ( d, N/3, MPI_INT, 2, 0, MPI_COMM_WORLD );
        }
```

```
	else{
		if( my_id == 1  )
			MPI_Recv ( c, N/3, MPI_INT, 0, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE );
		else
			MPI_Recv ( d, N/3, MPI_INT, 0, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE );
	}

	/* se hacen las ordenaciones parciales*/
	if (  my_id == 0 ) {
		quickSort( b, 0, N/3 - 1);
	}
	else{
		if ( my_id == 1)
			quickSort( c, 0, N/3 - 1);
		else
			quickSort( d, 0, N/3 - 1);
	}

	/* se envia el resultado de las ordenaciones parciales a la
placa principal */
	if ( my_id == 1 ){
		 MPI_Send ( c, N/3, MPI_INT, 0, 0, MPI_COMM_WORLD );
	 }
	else if ( my_id == 2 ) {
		MPI_Send ( d, N/3, MPI_INT, 0, 0, MPI_COMM_WORLD);
	}
	 else{
		 MPI_Recv ( c, N/3, MPI_INT, 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE );
		MPI_Recv ( d, N/3, MPI_INT, 2, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE );
	}

	/* Funcion merge que imprime el resultado final ya
ordenado*/

	if ( my_id == 0){

		int MAX_B = 0, MAX_C = 0, MAX_D = 0, aux = 0;
		for(int i = 0; i < N; i++){

			if (MAX_B < N/3){
				if (MAX_C < N/3) {
					if (MAX_D < N/3){
						if (b[MAX_B] <
c[MAX_C]){
							if(d[MAX_B] <
d[MAX_D]){
								printf("
%d ", b[MAX_B]);
								MAX_B++;
							}
							else {
								printf("
%d ", d[MAX_D]);
```

```
MAX_D++;
                                                        }
                                                }
                                                else{
                                                        if (c[MAX_C] <
d[MAX_D]){
                                                                printf("
%d ", c[MAX_C]);

MAX_C++;
                                                        }
                                                        else{
                                                                printf("
%d ", d[MAX_D]);

MAX_D++;
                                                        }
                                                }
                                        }
                                        else{
                                                if(d[MAX_B] <
c[MAX_C]){
                                                                printf("
%d ", b[MAX_B]);
                                                                 MAX_B++;
                                                         }
                                                          else {
                                                              printf(" %d
", c[MAX_C]);

MAX_C++;
                                                    }
                                        }
                                }
                                else {
                                        if (MAX_D < N/3){
                                                if(d[MAX_B] <
d[MAX_D]){
                                                                printf(" %d
", b[MAX_B]);
                                                                MAX_B++;
                                                     }
                                                     else {
                                                                printf(" %d
", d[MAX_D]);
                                                                MAX_D++;
                                                     }
                                        }
                                        else{
                                                printf(" %d ",
b[MAX_B]);
                                                     MAX_B++;
                                        }
                                }
                        }
                        else{
                                if ( MAX_C < N/3){
```

```
                                        if (c[MAX_C] < d[MAX_D]){
                                                printf(" %d ",
c[MAX_C]);
                                                MAX_C++;
                                            }
                                            else{
                                                printf(" %d ",
d[MAX_D]);
                                                MAX_D++;
                                            }
                                    }
                                    else{
                                        printf(" %d ", d[MAX_D]);
                                            MAX_D++;
                                    }
                            }
                    }
            }

            /* se calcula el tiempo de fin */

            double aux = (((double) clock() - start) / CLOCKS_PER_SEC);

            /*se imprime el tiempo empleado en ordenar un array de
tamaño N */

            if ( my_id == 0  ){
                    printf("\n");
                    printf("Tiempo de ejecucion quicksort en MPI: %f ",
aux,"\n");
                    printf("\n");
            }

            MPI_Finalize();
            return 0;
}

void quickSort( int a[], int l, int r)
{
   int j;

   if( l < r )
   {
       j = partition( a, l, r);
       quickSort( a, l, j-1);
       quickSort( a, j+1, r);
   }
}

int partition( int a[], int l, int r) {
   int pivot, i, j, t;
   pivot = a[l];
   i = l; j = r+1;

   while(1)
   {
       do ++i; while( a[i] <= pivot && i <= r );
       do --j; while( a[j] > pivot );
```

```
        if( i >= j ) break;
        t = a[i]; a[i] = a[j]; a[j] = t;
    }
    t = a[l]; a[l] = a[j]; a[j] = t;
    return j;
}
```

Anexo G algoritmo multiplicación de matrices secuencial

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 256

int main(int argc, char *argv[]){

srand(time(NULL));

/*se declaran las matrices*/

int  **b, **c, **d;

b = (int**)malloc(N*sizeof(int));
c = (int**)malloc(N*sizeof(int));
d = (int**)malloc(N*sizeof(int));

int m;

for ( m = 0; m < N; m++){
       b[m] = (int*)malloc(N*sizeof(int));
       c[m] = (int*)malloc(N*sizeof(int));
       d[m] = (int*)malloc(N*sizeof(int));
}

/* se inicializan con números aleatorios */

for (int i=0; i<(N); i++)
   for (int j=0; j<(N); j++){
      b[i][j] = rand();
      c[i][j] = rand();
      d[i][j] = rand();
}

/* se guarda el tiempo de comienzo del algoritmo */
clock_t start = clock();

/* Se realiza la multiplicacion de matrices */

for(int i=0;i<N;i++)
   for(int j=0;j<N;j++)
      for(int k=0;k<N;k++){
       c[i][j]+= d[i][k] * b[k][j];
}

/* se mida el tiempo transcurrido*/
double aux = ((double)clock() - start) / CLOCKS_PER_SEC;;

/* se imprime el tiempo invertido para calcular la nueva matriz*/

printf("Tiempo en calcular la multiplicacion de matrices con 1
placa:");
printf(" %f",aux,"\n");
printf("\n");
```

```
/* se imprime resultado de la matriz */

printf("Resultado de la matriz:");
for (int i=0; i<N; i++) {
   printf("\n");
   for (int j=0; j<N; j++)
      printf("%i   ", c[i][j]);
   }
printf("\n");
}
```

Anexo H algoritmo multiplicación de matrices con MPI

```
#include<stdio.h>
#include<mpi.h>
#include <time.h>
#include <stdlib.h>

#define N 1024

#define MASTER_TO_SLAVE_TAG 1
#define SLAVE_TO_MASTER_TAG 4

void generaMatrizAB();
void Imprime();

int rank;
int size;
int i, j, k;


double matrizA[N][N];
double matrizB[N][N];
double matrizResultAB[N][N];


double start_time;
double end_time;


int LimiteInferiorParaPlacasEsclavas;
int LimiteSuperiorParaPlacasEsclavas;
int numFilasPorPlacasEsclavas;


MPI_Status status;
MPI_Request request;


int main(int argc, char *argv[])
{

    /* inicializacion de MPI */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /* Comenzamos a medir tiempo */
    clock_t start = clock();
    srand (time(NULL));

    /* Generamos los array a y b para repartirlos entre las
diferentes placas*/
    if (rank == 0) {
        generaMatrizAB();

        for (i = 1; i < size; i++) {
            numFilasPorPlacasEsclavas = (N / (size - 1));
```

```
            LimiteInferiorParaPlacasEsclavas = (i - 1) *
numFilasPorPlacasEsclavas;
           if (((i + 1) == size) && ((N % (size - 1)) != 0)) {
                LimiteSuperiorParaPlacasEsclavas = N;
            } else {
                LimiteSuperiorParaPlacasEsclavas =
LimiteInferiorParaPlacasEsclavas + numFilasPorPlacasEsclavas;
            }
            MPI_Isend(&LimiteInferiorParaPlacasEsclavas, 1,
MPI_INT, i, MASTER_TO_SLAVE_TAG, MPI_COMM_WORLD, &request);
            MPI_Isend(&LimiteSuperiorParaPlacasEsclavas, 1,
MPI_INT, i, MASTER_TO_SLAVE_TAG + 1, MPI_COMM_WORLD, &request);
           MPI_Isend(&matrizA[LimiteInferiorParaPlacasEsclavas][0],
(LimiteSuperiorParaPlacasEsclavas -
LimiteInferiorParaPlacasEsclavas) * N, MPI_DOUBLE, i,
MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD, &request);
        }
    }

    MPI_Bcast(&matrizB, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);


    /* las placas esclavas realizan su calculo */
    if (rank > 0) {
        /* receive low bound from the master */
        MPI_Recv(&LimiteInferiorParaPlacasEsclavas, 1, MPI_INT, 0,
MASTER_TO_SLAVE_TAG, MPI_COMM_WORLD, &status);
       MPI_Recv(&LimiteSuperiorParaPlacasEsclavas, 1, MPI_INT, 0,
MASTER_TO_SLAVE_TAG + 1, MPI_COMM_WORLD, &status);
       MPI_Recv(&matrizA[LimiteInferiorParaPlacasEsclavas][0],
(LimiteSuperiorParaPlacasEsclavas -
LimiteInferiorParaPlacasEsclavas) * N, MPI_DOUBLE, 0,
MASTER_TO_SLAVE_TAG + 2, MPI_COMM_WORLD, &status);
        for (i = LimiteInferiorParaPlacasEsclavas; i <
LimiteSuperiorParaPlacasEsclavas; i++) {
          for (j = 0; j < N; j++) {
              for (k = 0; k < N; k++) {
                      matrizResultAB[i][j] += (matrizA[i][k] *
matrizB[k][j]);
                }
            }
        }

       MPI_Isend(&LimiteInferiorParaPlacasEsclavas, 1, MPI_INT, 0,
SLAVE_TO_MASTER_TAG, MPI_COMM_WORLD, &request);
       MPI_Isend(&LimiteSuperiorParaPlacasEsclavas, 1, MPI_INT, 0,
SLAVE_TO_MASTER_TAG + 1, MPI_COMM_WORLD, &request);
       MPI_Isend(&matrizResultAB[LimiteInferiorParaPlacasEsclavas][
0], (LimiteSuperiorParaPlacasEsclavas -
LimiteInferiorParaPlacasEsclavas) * N, MPI_DOUBLE, 0,
SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD, &request);
}

       /* recibimos resultados de las placas esclavas*/
       if (rank == 0) {
        for (i = 1; i < size; i++) {
              MPI_Recv(&LimiteInferiorParaPlacasEsclavas, 1,
MPI_INT, i, SLAVE_TO_MASTER_TAG, MPI_COMM_WORLD, &status);
```

```
                MPI_Recv(&LimiteSuperiorParaPlacasEsclavas, 1,
MPI_INT, i, SLAVE_TO_MASTER_TAG + 1, MPI_COMM_WORLD, &status);

MPI_Recv(&matrizResultAB[LimiteInferiorParaPlacasEsclavas][0],
(LimiteSuperiorParaPlacasEsclavas -
LimiteInferiorParaPlacasEsclavas) * N, MPI_DOUBLE, i,
SLAVE_TO_MASTER_TAG + 2, MPI_COMM_WORLD, &status);
         }


        /* imprimimos resultados obtenidos */
         printf("\n Tiempo para calcular multiplicacion de matrices
con MPI = %f\n\n", end_time - start_time);
         Imprime();
    }

   /* imprimimos tiempo invertido */
   if (rank == 0) {
         double aux = ((double)clock() - start) / CLOCKS_PER_SEC;;
         printf("%.16g segundos\n", aux);
   }


    MPI_Finalize();
    return 0;
}

void generaMatrizAB()
{
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            matrizA[i][j] = rand();
        }
    }
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            matrizB[i][j] = rand();
        }
    }
}


void Imprime()
{
    for (i = 0; i < N; i++) {
        printf("\n");
        for (j = 0; j < N; j++)
            printf("%8.2f  ", matrizA[i][j]);
    }
    printf("\n\n\n");
    for (i = 0; i < N; i++) {
        printf("\n");
        for (j = 0; j < N; j++)
            printf("%8.2f  ", matrizB[i][j]);
    }
    printf("\n\n\n");
    for (i = 0; i < N; i++) {
        printf("\n");
```

```
        for (j = 0; j < N; j++)
            printf("%8.2f  ", matrizResultAB[i][j]);
    }
    printf("\n\n");
}
```

Anexo I algoritmo de Montecarlo secuencial

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>

int main(int argc, char** argv)
{
   unsigned long long int iteraciones=20000000;
   double x,y;
   int i,iteUsadas=0; /* # Numero de puntos dentro del 1 cuadrante
del circulo */
   double z;
   double pi;


   /* inicializamos los numeros aleatorios */
   srand(time(NULL));
   iteUsadas=0;

   /* declaramos las variables para medir el tiempo */
   struct timeval start, end;
   gettimeofday(&start,NULL);

   /* se realizan los calculos de pi, se evalua que entren en el 1
cuadrante del circulo */
   for ( i=0; i<iteraciones; i++) {
      x = (double)rand()/RAND_MAX;
      y = (double)rand()/RAND_MAX;
      z = x*x+y*y;
      if (z<=1) iteUsadas++;
   }

   /* se mide cuando termina la ejecucion del algoritmo*/
   gettimeofday(&end,NULL);

   /*se ve la diferencia entre el tiempo de comienzo y fin*/
   double aux = (end.tv_sec - start.tv_sec) + ((end.tv_usec -
start.tv_usec)/1000000.0);

   /* multiplicamos por 4 para obtener el valor de pi*/
   pi=(double)iteUsadas/iteraciones*4;

   /* imprimimos resultados */
   printf("pi     =  %g \n",pi);
   printf("Tiempo = %g \n",aux);

   return 0;
}
```

Anexo J algoritmo de Montecarlo con MPI

```
# include <math.h>
# include </usr/include/openmpi-x86_64/mpi.h>
# include <stdio.h>
# include <time.h>
#include <stdlib.h>

# define DEBUG             0
# define Iteraciones       2000000

/*
 *   Mensajes de control
 *   */
# define NecesitamosNumeros      1
# define NumeroAleatorio    2

int main ( int argc, char *argv[] );


/**********************************************************************
************/

int main ( int argc, char *argv[] ){
  double PI;
  int completado;
  int i;
  int ierr;
  int in;
  int max;
  MPI_Status estadoMPI;
  int my_id;
  int numProcesos;
  int out;
  int puntosMaximos = 10000;
  int placasEsclavas;
  int *numAleatorios = (int*)malloc(Iteraciones*sizeof(int));
  int ranks[1];
  int parar;
  int temp;
  int numTotalesDentroSemiCirculo;
  int numTotalesFueraSemiCirculo;
  MPI_Group worker_group;
  MPI_Comm workers;
  MPI_Group world_group;
  double x;
  double y;

  srand (time(NULL));


/*
 *   se inicializa MPI
 *   */
  ierr = MPI_Init ( &argc, &argv );

/*
```

```
 *   obtenemos el numero de placas.
 *   */
  ierr = MPI_Comm_size ( MPI_COMM_WORLD, &numProcesos );

/*
 *   se obtiene el id del proceso principal.
 *   */
  ierr = MPI_Comm_rank ( MPI_COMM_WORLD, &my_id );


  if ( my_id == 0 )
  {
    printf ( "MONTE_CARLO - Proceso principal:\n" );
    printf ( "  Estimacion de pi por el metodo de montecarlo con
MPI.\n" );
    printf ( "  El numero de proceso es:  %d.\n", numProcesos );

  }

  ierr = MPI_Comm_group ( MPI_COMM_WORLD, &world_group );
/*
 *   definimos el grupo de placas esclavas para los cálculos
 *
 *     */
  placasEsclavas = numProcesos-1;
  ranks[0] = placasEsclavas;
  ierr = MPI_Group_excl ( world_group, 1, ranks, &worker_group );

/*
 *se define el grupo de placas esclavas
 * */

  ierr = MPI_Comm_create ( MPI_COMM_WORLD, worker_group, &workers
);

/*
 *Se define el grupo de placas que estan libres para hacer los
calculos
 * */

  ierr = MPI_Group_free ( &worker_group );
/*
 *   comienza proceso de calculo
 *   */

/*
 *   Se evalua que sea la placa principal y se ve mide el tiempo
 *   */
  if ( my_id == placasEsclavas )
  {
    struct timeval time;
    gettimeofday( &time, 0 );
/*
 *   Inicializamos la generacion de numeros aleatorios
 *   */
    srandom ( (int)(time.tv_usec*1000000+time.tv_sec) );

    do
```

```
    {
      ierr = MPI_Recv ( &parar, 1, MPI_INT, MPI_ANY_SOURCE,
NecesitamosNumeros, MPI_COMM_WORLD, &estadoMPI );
      if ( parar )
      {
        for ( i = 0; i < Iteraciones; i++)
        {
          numAleatorios[i] = random();
        }
        ierr = MPI_Send ( numAleatorios, Iteraciones, MPI_INT,
estadoMPI.MPI_SOURCE, NumeroAleatorio, MPI_COMM_WORLD );
      }
    } while ( 0 < parar );

  }
/*
 *   para las placas esclavas
 *   */
  else
  {
    parar = 1;
    completado = in = out = 0;
    max = 2147483647;

    ierr = MPI_Send ( &parar, 1, MPI_INT, placasEsclavas,
NecesitamosNumeros, MPI_COMM_WORLD );
/*
 *   Generamos numeros aleatorios.
 *   */
    while (!completado)
    {
      parar = 1;
      ierr = MPI_Recv ( numAleatorios, Iteraciones, MPI_INT,
placasEsclavas, NumeroAleatorio, MPI_COMM_WORLD, &estadoMPI );

      for ( i = 0; i < Iteraciones; )
      {
        x = ( ( float ) numAleatorios[i++] ) / max;
        y = ( ( float ) numAleatorios[i++] ) / max;

        if ( x * x + y * y < 1.0 ) {
          in++;
        }
        else {
          out++;
        }

      }

/*
 *   numero de puntos dentro
 *   */
      temp = in;
      ierr = MPI_Reduce ( &temp, &numTotalesDentroSemiCirculo, 1,
MPI_INT, MPI_SUM, 0, workers );


/*
```

```
 *   numero de puntos fuera
 *   */
      temp = out;
      ierr = MPI_Reduce ( &temp, &numTotalesFueraSemiCirculo, 1,
MPI_INT, MPI_SUM, 0, workers );

      if ( my_id == 0 )
      {
        PI = ( 4.0 * numTotalesDentroSemiCirculo ) / (
numTotalesDentroSemiCirculo + numTotalesFueraSemiCirculo );
        completado =  puntosMaximos <= (
numTotalesDentroSemiCirculo + numTotalesFueraSemiCirculo );

        if ( completado ){
          parar = 0;
        }
        else{
          parar = 1;
        }

        ierr = MPI_Send ( &parar, 1, MPI_INT, placasEsclavas,
NecesitamosNumeros, MPI_COMM_WORLD );

        ierr = MPI_Bcast ( &completado, 1, MPI_INT, 0, workers );
      }
      else
      {
        ierr = MPI_Bcast ( &completado, 1, MPI_INT, 0, workers );

        if ( !completado )
        {
          parar = 1;
          ierr = MPI_Send ( &parar, 1, MPI_INT, placasEsclavas,
NecesitamosNumeros, MPI_COMM_WORLD );
        }
      }
    }
  }

/*
 *   Termina la ejecucion de MPI.
 *   */
  ierr = MPI_Finalize();

  if ( my_id == 0 )
  {
    printf ( "\n" );
    printf ( "MONTE_CARLO - Placa principal:\n" );
    printf( "pi = %23.20lf\n", PI );
  }
  return 0;
}
```

Bibliografía

[1] Abraham Silberschatz, G. G. *Operating System Concepts*. Obtenido de http://www-etud.iro.umontreal.ca/~dift2245/notes/2013/ch4.pdf (2013)

[2] Alonso, J. M. *Programación de aplicaciones paralelas.* Obtenido de http://www.sc.ehu.es/acwmialj/edumat/mpi.pdf (13 de Enero de 1997)

[3] Dalian, L. P. Research of parallel program performance analysis and optimization on multicore platform in Department of Compute science an technologie, Dalian Neusoft University of information. *Control Engineeringand information systems*. (2015)

[4] Datasheet, I.G. Obtenido de http://www.intel.com/content/dam/support/us/en/documents/galileo/sb/galileo_datasheet_329681_003.pdf (s.f.)

[5] Debian., P. Obtenido de https://www.debian.org/ (s.f.)

[6] El-Nashar, A. I. PARALLEL PERFORMANCE OF MPI SORTING ALGORITHMS ON DUAL–CORE PROCESSOR WINDOWS-BASED SYSTEMS . *International Journal of Distributed and Parallel Systems (IJDPS)*, Vol.2, No.3 PP 1-14 (May 2011).

[7] Fries, O. V. *Internet of things*. Obtenido de http://www.internet-of-things-research.eu/pdf/IoT-From%20Research%20and%20Innovation%20to%20Market%20Deployment_IERC_Cluster_eBook_978-87-93102-95-8_P.pdf (s.f.).

[8] HOEGER, H. &. *INTRODUCCION A MPI* . Obtenido de http://webdelprofesor.ula.ve/ingenieria/hhoeger/Introduccion_MPI.pdf (s/f).

[9] Kataria, P. *Parallel Quicksort Implementation Using MPI and Pthreads*. Technical report. RUID-117004233 (Diciembre de 2008).

[10] López, A. T. *Seguridad en el Internet de las Cosas, sección 2.1*. Obtenido de http://www.cait.upm.es/vigilancia_tecnologica/pluginfile.php/228/mod_resource

/content/2/Seguridad%20Internet%20de%20las%20Cosas%20(versi%C3%B3n%20Final).pdf (S/F).

[11] Mas, R. H. *El manual del Administrador de Debian*. Obtenido de https://debian-handbook.info/download/es-ES/stable/debian-handbook.pdf (2012-2015).

[12] *Open MPI*. Obtenido de https://www.open-mpi.org/ (s.f.).

[13] *OpenMP* . Obtenido de http://www.openmp.org/ (s.f.).

[14] Poonam Dabas, A. A. Grid Computing: An Introduction International Journal of Advanced Research in Computer Science and Software Enginiering. *Volume 3, Issue 3,*. (March 2013).

[15] Radenski, A. *Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs.* Obtenido de http://digitalcommons.chapman.edu/cgi/viewcontent.cgi?article=1017&context=scs_books (2011).

[16] S/A. *Bash Reference Manual*. Obtenido de https://www.gnu.org/software/bash/manual/bashref.html (2016)

[17] Says., G. *4.9 Billion Connected "Things" will be used in 2015*. Obtenido de http://www.gartner.com/newsroom/id/2905717 (2015)

[18] Sherihan Abu ElEnin, M. A. Evaluation of Matrix Multiplication on an MPI. I*nternational Journal of Electric & Computer Sciences IJECS-IJENS Vol: 11 No: 01 PP 50-57* (2011).

[19] Student IV SEM, M. T. (Volume: 02 Issue: 03). Home Automation Using Internet of Things. *International Research Journal of Engineering and Technology (IRJET)* June-2015.

[20] X*ingmin*g Zhou Stefan Jähnichen, M. X. Advanced Parrallel Processing Technologies. *Lecture Notes in Computer Science* (2003).

[21] Roldán Rafael Caballero, González Teresa Hortalá, Olier Narciso Martí, Soto Susana Nieva y Artalejo Mario Rodríguez, Matemática Discreta para Informáticos, *Pearson education.*

[22] Baro Elías y Tomeo Venacio, Introducción al Álgabra lineal, *Garceta*, página 48.

[23] S/A. https://www.ripe.net/

[24] Y.Y Teng & Z.G.HE, *Research of paralell program performance analysis and optimization on multicore platform in Deparment of Compute science an technologie* (2015).

[25] MPI_REDUCE (3) man page (version 2.0.3) (s.f.) obtenido de https://www.open-mpi.org/doc/v2.0/man3/MPI_Reduce.3.php